

معماری میکروسرویس

Microservice Architecture

چگونه با معماری میکروسرویس، اپلیکیشن‌های مُدرن بسازیم؟

نویسنده: علیرضا ارومند



nikamooz;
تجربه، آموزش، آینده

شما مجاز هستید!



این کتاب الکترونیکی را به همین شکل با دوستان خود
به اشتراک بگذارید و دانش را گسترش دهید.



nikamooz;
تجربه، آموزش، آینده



میکروسرویس

فیبای میکروسرویس /
سرشاسه: ارومید، علیرضا، ۱۳۶۴۰
عنوان و نام بدباد آور: میکروسرویس / نویسنده: علیرضا ارومید؛ برنامه‌ریزی و اجرا شرکت فناوری اطلاعات نیکآموز
مشخصات ظاهری: ۱۳۹۹، کلید آموزش، ۱۸۴ ص. مصور، جدول، نمودار، ۲۱/۵×۱۴/۵ س م
شابک: ۹۷۸-۶-۳۷۴-۵۲۵-۵
وضعیت فهرست نویسی: فیبا
موضوع: معماری خدمت‌گرا (کامپیوتر)
موضوع: Service-oriented architecture (Computer science)
موضوع: رایانش ابری
موضوع: Cloud computing
شناسه افزوده: شرکت فناوری اطلاعات نیکآموز
رده بنده کنگره: ۵۸۲۷/TK5105
رده بنده دیوبی: ۱۴۰۳۸۰۴۱۱/۶۵۸
شماره کتابشناسی ملی: ۷۵۱۸۷۴۹
وضعیت رکورد: فیبا



عنوان: میکروسرویس

نویسنده: علیرضا ارومید

برنامه‌ریزی و اجرا: شرکت فناوری نیکآموز

مدیر عامل: فرید طاهری

انتشارات: کلید آموزش

طراح جلد: سلما سرابی

صفحه‌آرا: حبیبه دربایی اصل

چاپ اول: ۱۳۹۹

تیراز: ۱۰۰ نسخه

قیمت: رایگان

شابک: ۹۷۸-۶-۳۷۴-۵۲۵-۵



درباره نویسنده

علیرضا ارومند

علیرضا ارومند به عنوان Product Manager در شرکت داتین (وابسته به فناپ) در حوزه پروژه‌های بانکی و بیمه‌ای فعال است. ایشان یکی از افراد پیشرو در حوزه‌های مهندسی نرم‌افزار و .NET Core در ایران است.

علیرضا ارومند همچنین مدرس بنام و Technical Manager پروژه‌های نیکآموز بوده و با مدیریت این تیم، پروژه‌های بسیار بزرگی را با موفقیت در نیکآموز انجام داده است.

- Product Manager شرکت داتین
- معاونت فنی شرکت راهکارهای بیمه‌گری مدرن داتین
- مشاور و معمار ارشد نرم‌افزار
- مدرس دوره‌های ASP.NET Core و مهندسی نرم‌افزار در نیکآموز
- همکاری با تیم توسعه شرکت ارتباط فردا (وابسته به بانک آینده)
- مشاور فنی شرکت توسعه رفاه پرديس (بانک رفاه)



اطلاعات بیشتر درباره مدرس

پیشگفتار

دنیای نرم افزار به سرعت در حال گسترش است و شرکت های بسیاری کسب و کارهای خود را بر اساس نرم افزار بنا نهاده اند. با توجه با بازار کسب و کار و نیاز شرکت ها به رقابت با یکدیگر، لازم است نرم افزارها توانایی رقابت را ایجاد کنند. برای داشتن نرم افزاری قابل رقابت باید چندین ویژگی مختلف در هر نرم افزاری وجود داشته باشد. نرم افزارها باید قابلیت تغییر سریع داشته باشند، پایداری بالایی داشته و امکان پاسخ به تعداد زیادی درخواست را دارا باشند و با توجه به ماهیت نرم افزارها که دیگر ابزارهای جانبی نیستند و بنیان کسب و کارها را تشکیل می دهند باید امنیت بالایی نیز داشته باشند.

در چنین شرایطی همه ارکان توسعه نرم افزار تحت تاثیر قرار می گیرند. یکی از این ارکان اساسی معماری نرم افزار است. معماری سنتی و یکپارچه گذشته دیگر توانایی همراهی با این سرعت تغییرات را دارا نیست. به همین دلیل در حال حاضر معماری میکروسرویس به عنوان یکی از به روزترین روش های توسعه نرم افزار توسط بسیاری از شرکت های بزرگ نرم افزاری مورد استفاده قرار می گیرد. اما در این شرایط چالش های جدیدی به دنیای توسعه نرم افزار وارد می شود. چالش هایی از قبیل تعیین حوزه هر سرویس، مدیریت داده های توزیع شده، برقراری امنیت در معماری جدید و استفاده از ابزارها و روش های جدید بخشی از این پیچیدگی ها و چالش ها است که باید به آن ها جواب داده شود. در ادامه در این کتاب سعی شده است که این چالش ها پاسخ داده

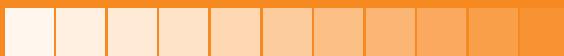
فهرست:

۱۴	فصل اول – میکروسرویس چیست؟
۱۴	دنبال قبیل از میکروسرویس به چه صورت بود؟
۱۶	پیش به سوی جهنم!
۱۹	بهشت گمشده میکروسرویس
۲۲	مزایای میکروسرویس‌ها
۲۴	معایب میکروسرویس‌ها
۳۰	فصل دوم – آشنایی با API Gateway
۳۳	تعامل مستقیم Client‌ها با میکروسرویس‌ها
۳۵	حل مشکلات با استفاده از API Gateway
۴۴	فصل سوم – ارتباط بین سرویس‌ها
۴۵	انواع روش‌های تعامل
۴۸	تعریف API‌ها
۴۹	تکامل API‌ها
۵۱	مدیریت بحران هنگامی که بخشی از سیستم دچار مشکل می‌شود
۵۶	فصل چهارم – تکنولوژی‌های ارتباطی
۵۷	ارسال پیام به صورت Async
۶۱	ارتباط Request/Response و Sync
۶۱	سرویس‌های REST
۶۲	مزایای HTTP
۶۳	معایب HTTP
۶۳	ساختار پیام‌ها

۶۶	فصل پنجم - آشنایی با Service Discovery
۶۸	آشنایی با Service Registry
۶۸	آشنایی با الگوی Client-Side Discovery
۷۰	آشنایی با الگوی Server-Side Discovery
۷۲	بررسی دقیق Service Registry
۷۴	انواع روش‌های مدیریت سرویس‌ها
۸۰	فصل ششم - مدیریت داده‌ها در میکروسرویس‌ها
۸۰	مشکلات داده‌های توزیع شده در میکروسرویس‌ها
۸۵	توسعه بر مبانی Event‌ها
۹۰	ویژگی Automicity هنگام استفاده از Event‌ها
۹۸	فصل هفتم - امنیت در میکروسرویس
۱۰۰	تاریخچه، نحوه امن کردن نرم‌افزارهای پکیارچه
۱۰۴	مبانی کلیدی امنیت
۱۱۲	امنیت مرزهای نرم‌افزار
۱۱۹	امن کردن ارتباط سرویس به سرویس
۱۳۶	فصل هشتم - آشنایی با روش‌های انتشار
۱۳۷	نصب چند سرویس مختلف به ازای هر میزبان
۱۳۹	نصب یک سرویس به ازای هر میزبان
۱۳۹	نصب هر سرویس روی یک ماشین مجازی
۱۴۱	نصب و راهاندازی به کمک کانتینرها
۱۴۶	فصل نهم - مهاجرت به میکروسرویس
۱۴۸	کار را متوقف کنید

۱۵۱	جداسازی Back-end و Front-end
۱۵۳	استخراج سرویس
فصل دهم – آشنایی با میکروفرانت اند	
۱۶۰	سیستم‌ها و تیم‌ها
۱۶۳	خروجی کار Front-end
۱۶۸	ادغام Integration
۱۷۱	موضوعات مشترک
فصل یازدهم – مشکلاتی که حل می‌کنیم	
۱۷۴	سرعت بالا در افزودن ویژگی‌های جدید
۱۷۶	حذف گلوگاه Front-end
۱۷۸	توانایی تغییر
۱۷۹	مزایایی عدم وابستگی
۱۸۱	معایب Micro Front-end

فصل اول: میکروسرویس چیست؟



- دنیا قبل از میکروسرویس به چه صورت بود؟
- پیش به سوی جهنم!
- بهشت گمشده میکروسرویس [Microservice]
- مزایای میکروسرویس‌ها
- معایب میکروسرویس‌ها
- جمع‌بندی [میکروسرویس‌ها]

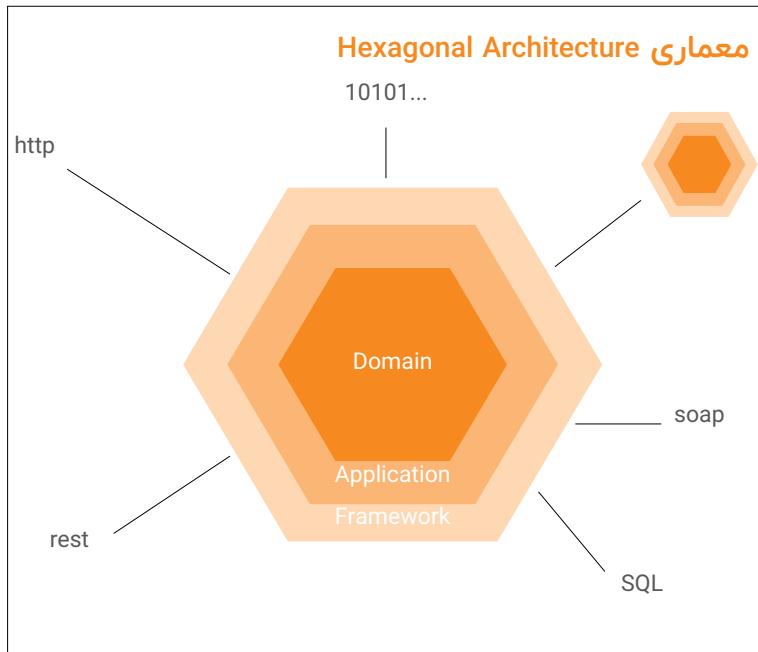
دنیا قبل از میکروسرویس به چه صورت بود؟

فرض کنید شما برای توسعه یک نرم افزار دعوت به همکاری شده اید. از شما خواسته می شود یک CMS خبری توسعه بدھید که کارش بسیار ساده است. خبرنگارها امکان ثبت اخبار متنی دارند، دبیرها و سردبیرها امکان انتخاب و انتشار اخبار در صفحات سایت دارند و در نهایت کاربران سایت امکان مشاهده مطالب سایت را خواهند داشت.

از آنجا که شما یک توسعه دهنده حرفه‌ای هستید احتمالاً سراغ یک ابزار خوب و یک ساختار خوب و تمیز خواهید رفت، مثلاً معماری Hexagonal یا معماری Onion برای توسعه انتخاب می‌کنید و در نهایت یک نرم افزار بسیار تمیز توسعه می‌دهید. در همچین شرایطی احتمالاً منطق برنامه در مرکز برنامه قرار گرفته و جاهایی که نیاز به ارتباط با زیرساخت و استفاده از ابزار دارید به جای وابسته شدن به تکنولوژی و زیرساخت از اینترفیس‌ها استفاده می‌کنید و وابستگی‌ها را به خارج از دامنه اصلی برنامه هدایت می‌کنید خروجی مناسب برای برنامه خودتان طراحی و پیاده‌سازی می‌کنید.

هر احمدقی می‌تونه کدهایی بنویسه که کامپیوتر

بفهمه اما برنامه‌نویس خوب کدهایی می‌نویسه که دیگر
انسان‌ها هم بتونن درک کنن.
مارتین فاولر



هر چند در این روش از نظر منطقی برنامه ما کاملاً مازولار طراحی و پیاده‌سازی شده است اما در واقع کل این مازولها کاملاً وابسته به هم هستند و در قالب یک بسته نرم‌افزاری روی خروجی قرار می‌گیرند. اینکه این یک بسته نرم‌افزاری دقیقاً چه چیزی است بستگی به تکنولوژی‌های مورد استفاده شما دارد اما در محیط .NET. شما یک یا چند اسمبلی دارید که در نهایت به عنوان خروجی برنامه شما شناخته خواهند شد و هر جایی نیاز به نرم‌افزار داشته باشید کل این اسمبلی‌ها باید در کنار هم بسته‌بندی بشوند و خدمات خودشان را ارائه بدهند.

توسعه برنامه به این روش بسیار فراگیر و مرسوم است. البته این فراگیری دلیل خوبی دارد و آن سادگی در توسعه و نصب برنامه است. ابزارها و IDها به سادگی این قابلیت را در اختیار شما قرار می‌دهند که برنامه‌های خودتان را توسعه داده و نصب و راهاندازی کنید. با یک نصب ساده قابلیت تست کل برنامه را دارید و برای راهاندازی نسخه جدید برنامه فقط کافی است بسته‌های نرم‌افزاری خودتان را کپی کنید و همه چیز آماده است. این روش توسعه همان چیزی است که در اصطلاح ما به آن Monolith می‌گوییم.

پیش به سوی جهنم!

از قدیم گفتند: "هیچ ارزانی بی حکمت نیست" اگر بخواهیم به زبان برنامه‌نویسی ترجمه کنیم می‌شود "هیچ سادگی بدون محدودیت نیست". هر نرم‌افزار و پروژه‌ای در صورتی که موفق بشود قطعاً تمايل به رشد دارد و موفقیت بیشتر پروژه موجب رشد بیشتر پروژه خواهد شد. در نهایت بعد از مدتی پروژه کوچک و ساده ما تبدیل به هیولا‌بی بزرگ و وحشتناک خواهد شد.

اما ممکن است به این فکر کنید که زیاد شدن حجم کدها در طول دوران توسعه نرم‌افزار و تغییرات آن چه مشکلی می‌تواند داشته باشد؟ در ادامه به چند مورد از این مشکلات خواهیم پرداخت.

مشکل اول: با بزرگ و پیچیده شدن نرم‌افزار تیم توسعه شما با مشکلات فراوانی دست و پنجه نرم خواهد کرد. هر تصمیمی برای تغییر در برنامه یا ایجاد سریع یک ویژگی و ارائه آن احتمالاً به بن بست

خواهد رسید. بزرگترین مشکل این نرم‌افزارها پیچیدگی بیش از حد این برنامه‌ها است. معمولاً نرم‌افزارها در طول سالیان آنقدر بزرگ و پیچیده می‌شوند که درک درست و دقیق عملکرد آنها برای یک توسعه دهنده نرم‌افزار غیرممکن می‌شود.

در نتیجه این عدم توانایی شناخت درست و صحیح باعث می‌شود رفع خطاهای موجود یا اضافه کردن یک ویژگی جدید هم بسیار سخت و هم بسیار پیچیده باشد. نکته اصلی در این شرایط این است که با گذشت زمان این مشکل به شکل نمایی بیشتر و بیشتر می‌شود. هر چقدر فهم کد سخت‌تر بشود احتمال اشتباه بیشتر و احتمال پیدا کردن اشتباه‌ها کمتر و توان رفع آنها هم کمتر می‌شود، در نهایت به سورس کدی خواهیم رسید که اصطلاحاً به آن Big Ball of Mud می‌گوییم.

مشکل دوم: سورس کد حجیم می‌تواند باعث پایین آمدن سرعت توسعه نرم‌افزار بشود. هر وقت تصمیم به باز کردن پروژه بگیرید دقایق زیادی طول خواهد کشید تا IDE شما باز بشود و آماده به کار باشد. در کنار این شرایط احتمالاً این سیستم عظیم بهره‌وری کل سیستم شما را هم پایین خواهد آورد.

مشکل سوم: احتمالاً با داشتن یک سیستم بزرگ شما با عدم توانایی در انتشار بهبودها و توسعه‌های کوچک رو برو خواهید شد. مسلماً سیستمی که باز کردن آن چند دقیقه طول بکشد، Build شدنش بعضاً ۱ ساعت طول خواهد کشید و روال نصب راحتی هم نخواهد داشت. همچنین با توجه به اینکه در زمان نصب احتمالاً سیستم از دسترس

خارج خواهد بود و قاعده‌تاً از دسترس خارج کردن یک سیستم عظیم به خاطر یک تغییر کوچک یا رفع باگ احتمالی جزئی، مقرر نه به صرفه نیست به همین خاطر نصب نسخه‌های جدید با فاصله‌های زمانی طولانی انجام خواهد شد.

مشکل چهارم: عدم استفاده بهینه از منابع یکی دیگر از مشکلات اساسی توسعه نرم‌افزار به این شکل است. در نرم‌افزارهای متفاوت RAM نیازهای متفاوتی وجود دارد. ممکن است بخشی از برنامه مصرف زیادی داشته باشد و بخشی دیگر هم مصرف CPU اما در این روش امکان تخصیص یک منبع خاص به بخش خاصی که نیاز به آن منبع دارد، وجود نخواهد داشت. در نتیجه هنگامی که بخشی با مصرف CPU زیاد ارتقا داده بشود این امکان در اختیار همه بخش‌ها قرار خواهد گرفت و برای همه قسمت‌ها امکان استفاده از این منبع، بی‌رویه و ناصحیح وجود خواهد داشت.

مشکل پنجم: مشکلی که توسعه به روش Monolith به همراه داره انتشار مشکلات است. کل برنامه در یک سیستم و در قالب یک پروسه اجرا خواهد شد. پس اگر ایرادی در هر یک از قسمت‌های برنامه به وجود بیاید، تمامی قسمت‌های برنامه از کار خواهند افتاد و کل برنامه تا زمان رفع مشکل و نصب نسخه جدید از دسترس خارج خواهد بود. البته در این شرایط باید امیدوار بود که نصب نسخه جدید موجب ایجاد مشکلات جدید در سیستم نشود.

مشکل ششم: عدم توانایی در به کارگیری ابزارها و تکنولوژی‌های جدید

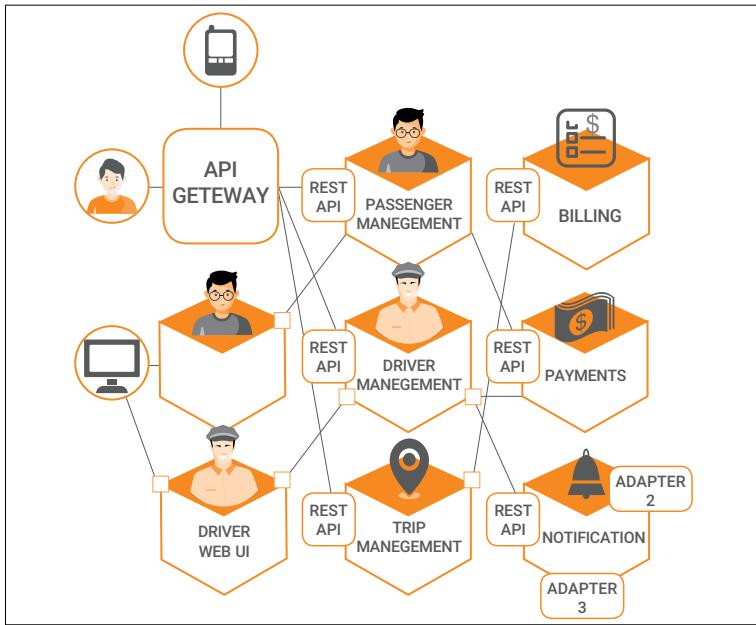
هم یکی دیگر از ایرادات این روش توسعه نرم‌افزار است. در شرایطی که شما یک نرم‌افزار بزرگ و پیچیده را سال‌ها توسعه داده‌اید احتمالاً وقتی با یک ابزار، تکنولوژی یا فریمورک جدید مواجه بشوید به جزء حسرت امکانات موجود کار دیگری از دستتان برnmی‌آید. معمولاً امکان اینکه سال‌ها تلاش تیم دور ریخته بشود نه پذیرفته می‌شود نه قابل اجرا هست. در حال حاضر در یکی از شرکت‌های بزرگ درگیر مشاوره در یک پروژه ERP هستم که سال‌ها پیش و با تکنولوژی سال ۲۰۰۵ توسعه داده شده است در این ابزار از .NET Framework 2.0 استفاده شده است و برای توسعه وب هم از ASP.NET Web Form استفاده شده است و اینقدر سیستم بزرگ و پیچیده شده که در طی این سال‌ها کسی جرات دست زدن به سیستم و تغییر تکنولوژی را نداشته است در صورتی که خودتان جای یکی از مدیران پروژه و شرکت بگذرید چقدر احتمال دارد قبول کنید که ثمره ۱۵ سال توسعه یک تیم برنامه نویسی دور ریخته بشود و یک پروژه جدید استارت زده شود؟ به نظرتان در این ۱۵ سال که یک تیم ۲۰-۱۵ نفره به طور میانگین درگیر این پروژه بوده است چقدر هزینه تولید همچین ابزاری شده است؟ آیا تغییر تکنولوژی با این شرایط امکان پذیر هست؟ آیا کم در حال نزدیک شدن به پایان دنیا هستیم. اما واقعاً راه حلی برای این مشکلات نیست؟

بهشت گمشده: میکروسرویس [Microservice]

بسیاری از شرکت‌های بزرگ نرم‌افزاری دنیا مثل EBay, Amazon, Net- flix, Facebook و ... برای حل این مشکل به سراغ توسعه به روش

میکروسرویس رفتند. این شرکت‌ها تصمیم گرفتند که به جای داشتن هیولای غیرقابل کنترل تعداد زیادی مینی‌اپلیکیشن تولید کنند که خیلی خوب با هم تبادل اطلاعات می‌کنند و هر کدام از این مینی‌اپلیکیشن‌ها یک وظیفه خاص و دقیق را به انجام می‌رسانند.

در این روش هر سرویس مجموعه‌ای از وظایف مرتبط با هم را به طور کامل و بدون وابستگی به بخش دیگری به انجام می‌رساند. برای مثال در سیستم تولید محتوا و مدیریت خبر می‌توان به مواردی چون مدیریت نظرات، مدیریت ثبت خبر و محتوا، مدیریت فایل، مدیریت انتشار در بستر وب و ... اشاره کرد. در این روش هر مینی‌اپلیکیشن از یک عماری تمیز مثل Onion یا Hexagonal به طور داخلی استفاده می‌کند. هر مینی‌اپلیکیشن این توانایی را خواهد داشت که در صورت نیاز بخشی از خدمات و داده‌های خودش را برای سایر قسمت‌ها به صورت API در اختیار قرار بدهد. برای نصب و راهاندازی هم هر کدام از این مینی‌اپلیکیشن‌ها توانایی این را خواهد داشت که در یک VM جداگانه به کار خودشان ادامه بدهند یا به عنوان Docker Image در اختیار تیم زیرساخت برای نصب و راهاندازی قرار بگیرند.



در این شرایط هر کدام از بخش‌های عملیاتی برنامه به عنوان یک مینی‌اپلیکیشن توسعه داده شدند. مهم تر از آن حالا به جای یک وب اپلیکیشن بزرگ مجموعه‌ای از اپلیکیشن‌های کوچک داریم که به طور دقیق و حساب شده‌ای برای انجام کار اصلی و رسیدن به هدف اصلی با هم تعامل و همکاری خواهند کرد. برای مثال در بخش تولید خبر این امکان برای این مینی‌اپلیکیشن فراهم هست که برای نمایش تصاویر از مینی‌اپلیکیشن مربوط به مدیریت فایل‌های عکس کمک بگیرد و از طریق API‌های ارائه شده توسط بخش مدیریت تصاویر به عکس‌های ذخیره شده در سیستم دسترسی داشته باشد. برای ارتباط بین سرویس‌ها راهکارهای مختلفی وجود دارد و اگر با این روش‌ها آشنا نیستید اصلاً نگران نباشید. در قسمت‌های بعد در مورد این روش‌ها کامل صحبت خواهیم کرد.

خوب تا اینجا همه چیز به نظر خوب می‌آید اما احتمالاً به این موضوع فکر بکنید که اگر تعداد مینی‌اپلیکیشن‌ها زیاد بشود و هر برنامه هم API خودش را ارائه بدهد و برنامه‌ها نیاز داشته باشند که با هم ارتباط برقرار کنند تعداد زیادی آدرس بوجود خواهد آمد که هر برنامه باید آنها را مدیریت کند و این خودش شروع مشکلات می‌شود. اما خبر خوب اینکه این مشکل به کمک API Gateway حل خواهد شد و در مورد جزئیات این بحث در قسمت آینده کامل صحبت خواهیم کرد.

مزایای میکروسرویس‌ها

گویا این روش توسعه جدید نرمافزار بسیاری از مشکلات روش Mono را حل خواهد کرد. اما برای اینکه دقیق‌تر بدانیم به چه شکلی این مشکلات حل خواهند شد بباید با هم نگاهی به برخی ویژگی‌های مفید میکروسرویس‌ها بندازیم.

احتمالاً همه شما با روش تقسیم و غلبه برای حل کردن مشکلات آشنا هستید، به جای حل کردن یک مشکل بزرگ بهتر است مشکل به قطعات کوچک شکسته بشود و هر قطعه به طور جداگانه حل بشود و در نهایت جواب نهایی از مجموع جواب‌های به دست آمده تشکیل خواهد شد. خوب همین فلسفه در مورد میکروسرویس‌ها هم صدق می‌کند. هیولای Monolith به تعداد زیادی مینی‌اپلیکیشن با قابلیت مدیریت و نگهداری بهتر شکسته می‌شود و حالا تعداد زیادی صورت مسئله جزئی برای حل کردن خواهیم داشت. قطعاً با کوچک شدن برنامه‌ها فهم و توسعه برنامه‌ها هم به شدت ساده‌تر از قبل خواهد شد.

اغلب شرکت‌های نرم‌افزاری از تعدد Stack‌های توسعه نرم‌افزار وحشت دارند و معمولاً محدودیت‌های زیادی در انتخاب ابزارها و فریم‌ورک‌های توسعه نرم‌افزار وجود دارد با این حالا در شرایطی که تعداد زیادی مینی‌اپلیکیشن داریم به راحتی می‌توانیم در هر کدام از این مینی‌اپلیکیشن‌ها از ابزارهایی که دقیقاً مناسب با نیاز آن‌ها است استفاده کنیم. به راحتی اگر داده‌های ما ساختار گراف داشته باشید به سراغ یک گراف دیتابیس می‌رویم. برای هر برنامه زبان توسعه خاص آن را انتخاب می‌کنیم و به راحتی به هر تکنولوژی و فریم ورکی سلام خواهیم کرد.

یکی دیگر از ویژگی‌های توسعه میکروسرویس قابلیت نصب و انتشار مستقل هر کدام از این میکروسرویس‌ها است. دیگر نیازی نیست برای رفع یک باگ کوچک در یک قسمت کوچک برنامه کل سیستم از دسترس خارج شود. بلکه به سادگی همان قسمتی که نیاز به رفع ایراد دارد در مدت کوتاهی راهاندازی مجدد خواهد شد.

اما استفاده بهینه و صحیح از منابع هم شاید یکی از مهم‌ترین ویژگی‌های توسعه میکروسرویس باشد. در این روش هر مینی‌اپلیکیشن به اندازه نیاز خود منابع و امکانات دریافت خواهد کرد و در صورت نیاز می‌توان منابع یک سرویس را افزایش داد یا یک سرویس خاص را در چند نسخه مختلف اجرا کرد.

خیلی هم خوب و خیلی هم عالی تا اینجای کار تمام معایب سیستم‌های قدیمی رفع شده و می‌توانیم به راحتی به کار خود ادامه دهیم. اما طبق اصل قطعاً به دست آوردن این همه مزیت بدون هزینه و ایراد خواهد بود. پس در ادامه بعضی از این ایرادات را با هم بررسی خواهیم کرد.

در این شرایط هر کدام از بخش‌های عملیاتی برنامه به عنوان یک مینی‌اپلیکیشن توسعه داده شدند. مهم‌تر از آن حالا به جای یک وب اپلیکیشن بزرگ مجموعه‌ای از اپلیکیشن‌های کوچک داریم که به طور دقیق و حساب شده‌ای برای انجام کار اصلی و رسیدن به هدف اصلی با هم تعامل و همکاری خواهند کرد. برای مثال در بخش تولید خبر این امکان برای این مینی‌اپلیکیشن فراهم هست که برای نمایش تصاویر از مینی‌اپلیکیشن مربوط به مدیریت فایل‌های عکس کمک بگیرد و از طریق API‌های ارائه شده توسط بخش مدیریت تصاویر به عکس‌های ذخیره شده در سیستم دسترسی داشته باشد. برای ارتباط بین سرویس‌ها راهکارهای مختلفی وجود دارد و اگر با این روش‌ها آشنا نیستید اصلاً نگران نباشید. در قسمت‌های بعد در مورد این روش‌ها کامل صحبت خواهیم کرد.

احتمالاً به این موضوع فکر بکنید که اگر تعداد مینی‌اپلیکیشن‌ها زیاد بشود و هر برنامه هم API خودش را ارائه بدهد و برنامه‌ها نیاز داشته باشند که با هم ارتباط برقرار کنند تعداد زیادی آدرس بوجود خواهد آمد که هر برنامه باید آنها را مدیریت کند و این خودش شروع مشکلات می‌شود. اما خبر خوب اینکه این مشکل به کمک API Gateway حل خواهد شد و در مورد جزئیات این بحث در قسمت آینده کامل صحبت خواهیم کرد.

معایب میکروسرویس‌ها

شاید بزرگترین ایراد توسعه به روش میکروسرویس نام این روش باشد. به طور جدی در این نام روی اندازه کوچک و یا بهتر بگوییم بسیار کوچک این سرویس‌ها تاکید شده است. اما این کوچک بودن به چه معناست؟

چقدر کوچک؟ اندازه یک مورچه به نسبت یک گربه بسیار کوچک است. اما همان گربه به نسبت یک فیل کوچک به حساب می‌آید و در مجموع به اندازه کل کره زمین به نسبت کل کائنات کمی فکر کنید بینید کدام یک از این مواردی که اسم بردمی کوچک به حساب می‌آید. پس تعیین مقیاس برای سرویس‌ها یکی از مشکلات ما خواهد بود. هنگامی که به اندازه سرویس‌ها فکر می‌کنید حتماً این نکته را به یاد داشته باشید که کوچک بودن اندازه سرویس‌ها وسیله‌ای برای رسیدن به هدفی بزرگ است نه یک هدف اصلی و بزرگ.

پیچیدگی برقراری ارتباط بین سرویس‌های مختلف و سایر پیچیدگی‌های فنی دیگری که هنگام توسعه یک سیستم توزیع شده با آن مواجه خواهیم شد بسیار بیشتر از زمانی است که یک نرم‌افزار Monolith توسعه می‌دهیم و قبل از انتخاب این روش توسعه، باید به این پیچیدگی‌ها و راهکارهایی که برای برخورد با این پیچیدگی‌ها داریم فکر کنیم. برای مثال برای استفاده از امکانات یک زیرسیستم دیگر در روش توسعه Monolith به راحتی از کلاسی نمونه‌سازی می‌کنیم و تابعی از آن کلاس را صدا می‌زنیم اما این کار را در یک سیستم توزیع شده نمی‌توانیم انجام دهیم.

در سیستم‌هایی که به روش Microservice توسعه می‌دهیم تاکید فراوانی بر توانایی عملکرد هر سیستم به تنها‌یی وجود دارد و این بدان معنا است که هر میکروسرویس دیتابیس اختصاصی خود و داده‌های اختصاصی خود را خواهد داشت و این توزیع شدگی داده‌ها در چند دیتابیس و مدیریت نسخه‌های موجود از یک داده در دیتابیس‌های مختلف می‌تواند مشکلات زیادی را برای تیم توسعه به وجود آورد. شاید این مشکل زمانی بیشتر به

چشم بخورد که اتفاقی در سیستم رخ دهد که نیاز به تغییر در پایگاه داده در چند سرویس مختلف داشته باشد و نیاز داشته باشیم یک Transaction را بین چند سرویس مختلف مدیریت کنیم.

یکی دیگر از شمشیرهای دولبه در روش توسعه میکروسرویس نصب و راهاندازی آن است. شاید اگر به چند پاراگراف بالاتر برگردید مشاهده کنید که امکان نصب و راهاندازی جداگانه سرویس‌ها را به عنوان یکی از برتری‌های این روش توسعه نرم‌افزار شمردیم. اما با کمی دقیق خواهید دید که همین مورد می‌تواند ایرادات زیادی را ایجاد کند. بعضاً ممکن است یک سیستم بزرگ به بیش از ۱۰۰ سرویس کوچک تقسیم شود و نصب و راهاندازی و تنظیم ارتباطات این ۱۰۰ سرویس می‌تواند بسیار زمان‌گیر و پیچیده و پرخطا باشد.

برای بسیاری از این مشکلات راهکارهای عملیاتی زیادی تدارک دیده شده‌است که در فصل‌های آتی بررسی خواهیم کرد.

جمع‌بندی [میکروسرویس‌ها]

در این فصل سعی کردیم با دو روش توسعه Monolith و میکروسرویس و مزایا و معایب هر کدام از این روش‌ها آشنا شویم. به عنوان یک توسعه دهنده با نزدیک به ۱۷ سال سابقه توسعه نرم‌افزارهای مختلف یک روحیه مشترک بین همه توسعه دهندها سراغ دارم و این روحیه علاقه به استفاده از روش‌ها و ابزارهای جدید بدون توجه به نیاز واقعی کار است. پس پیشنهاد می‌کنم به جای تصمیم به توسعه تمام برنامه‌ها به روش میکروسرویس، قبل از انتخاب این روش، کاملاً نیازهای اصلی سیستم را

بررسی کنید و هر کدام از این روش‌ها را که مناسب سناریوی شما است انتخاب کنید. هر چند میکروسرویس بسیار مطرح و پرطرفدار است اما با یک بررسی سریع می‌توانید نمونه‌های شکست خورده زیادی از سناریوهایی که برای توسعه میکروسرویس را انتخاب کرده‌اند پیدا کنید.

تا ۱۵ سال آینده همون طور که خوندن و نوشتن رو به بچه‌ها یاد می‌دیم، برنامه‌نویسی رو هم یاد خواهیم داد و افسوس می‌خوریم که چرا زودتر این کار رو شروع نکردیم.

مارک زاکربرگ

فصل دوم: آشنایی با API Gateway

- تتعامل مستقیماً Client‌ها با میکروسرویس‌ها
- حل مشکلات با استفاده از API Gateway

در قسمت اول از این مجموعه درباره میکروسرویس‌ها، مزایا و معایب آن‌ها و چگونگی تاثیرگذاریشان بر دنیای توسعه نرم‌افزار صحبت کردیم. با اینکه توسعه میکروسرویس‌ها پیچیدگی‌های زیادی به همراه دارد، اما تاثیرات شگرفی بر حل پیچیدگی‌های موجود در توسعه نرم‌افزارهای بزرگ داشته که باعث شده به عنوان یکی از محبوب‌ترین روش‌های توسعه نرم‌افزار طی چند سال اخیر به حساب بیاید. در این قسمت و چند قسمت آینده در مورد برخی پیچیدگی‌ها و راه‌حل‌های آن‌ها و الگوهای توسعه میکروسرویس‌ها صحبت خواهیم کرد.

هنگامی که میکروسرویس‌ها خود را توسعه می‌دهید یک مسئله مهم پیش روی شما قرار دارد. چگونه Client‌های شما با میکروسرویس‌های شما تعامل خواهند کرد؟ هنگامی که از روش Monolithic برای توسعه نرم‌افزارهای خود استفاده می‌کنید نهایتاً یک Endpoint خواهید داشت و در صورت نیاز همین یک Endpoint برای تقسیم بار روی چند سرور نصب می‌شود و به کمک یک Load Balancer بار روی این سرورها توزیع می‌شود. اما زمانیکه میکروسرویس توسعه می‌دهید مجموعه‌ای از Endpoint‌ها خواهید داشت که باید بتوانید با آن‌ها تعامل کنید. در این قسمت قصد داریم راجع به تاثیرات تعداد زیادی Endpoint داشتن و نحوه حل کردن این مشکل صحبت کنیم.

مقدمه:

فرض کنید شما در حال توسعه اپلیکیشن برای یک فروشگاه آنلاین هستید. احتمالاً صفحه‌ای خواهید داشت که جزئیات محصولات را نمایش خواهد داد. بیایید نگاهی به صفحه محصولات دیجی‌کالا بیاندازیم. در این صفحه به راحتی می‌توان چندین بخش را تشخیص داد. بخش اصلی هدف صفحه است که همان نمایش اطلاعات محصول است. اما در این صفحه چند قسمت دیگر نیز به چشم می‌خورد. مثل: فروشنده‌گان، نقد و بررسی، نظرات کاربران، پرسش و پاسخ، محصولات پیشنهادی و ... اگر کمی ریزبین باشید احتمالاً متوجه بخش‌هایی که منجا انداخته‌ام هم شده‌اید مثل اطلاعات سبد خرید.

حالا فرض کنید که نیاز داریم یک موبایل اپلیکیشن هم برای فروشگاه آنلاین خود طراحی کنیم و در نتیجه در موبایل اپلیکیشن خود نیز به صفحه‌ای برای نمایش جزئیات کالا نیاز داریم. (هر چند نمایش این حجم داده در یک صفحه موبایل بسیار زیاد است اما برای سادگی مثال شما به روی خودتان نیاورید).

هنگامی که از روش Monolith استفاده می‌کنیم موبایل اپ ما تمام این داده‌ها را با یک درخواست ساده مانند زیر به دست خواهد آورد:

```
Get api.digikala.ir/product/111
```

در صورتی که برنامه ما تک نسخه‌ای باشد که به سادگی اجرا خواهد شد و نتیجه را باز خواهد گردانید و در صورتی که نرم‌افزار ما پشت Load

قرار داشته باشد درخواست توسط Load Balancer مسیریابی شده و به بهترین نسخه اجرایی نرم‌افزار تحویل داده خواهد شد و بعد از عبور از لایه‌های مختلف نرم‌افزار به دیتابیس خواهد رسید و با اجرای کوئری روی چند جدول دیتابیس نتیجه دلخواه آماده شده و در اختیار ما قرار خواهد گرفت.

در طرف مقابل اگر از روش میکروسرویس استفاده کرده باشیم داده‌هایی که به آن‌ها در صفحه جزئیات محصول نیاز داریم در اختیار چندین سرویس مختلف قرار دارد. در اینجا به چند نمونه از این میکروسرویس‌های احتمالی اشاره خواهیم کرد.

■ **سرویس سبد خرید:** نمایش تعداد داده‌های موجود در سبد کالا

■ **سرویس محصولات:** نمایش اطلاعات اصلی محصول

■ **سرویس تامین کنندگان:** نمایش داده‌های مربوط به تامین کنندگان کالا و قیمت گذاری‌های هر کدام

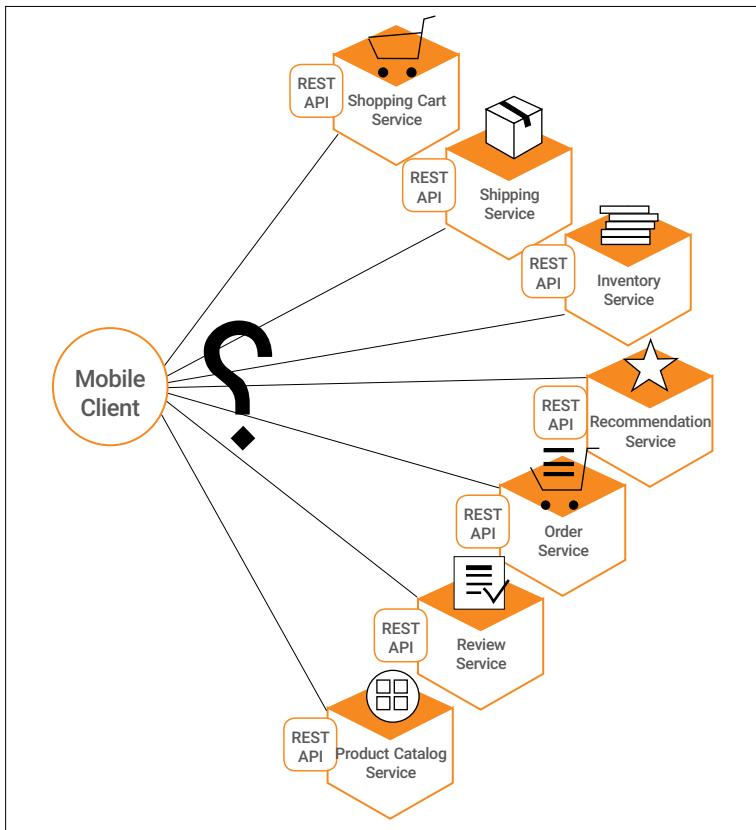
■ **سرویس نظرات:** دریافت و نمایش نظرات کاربران در مورد این محصول خاص

■ **سرویس پرسش و پاسخ:** برای دریافت سوالات و نمایش پاسخ سوالات موجود

■ **سرویس محصولات پشنهدادی:** بررسی عملکرد گذشته کاربر جاری و سایر کاربران و پیشنهاد کالا

حالا نیاز داریم تصمیم بگیریم اپلیکیشن موبایل ما چگونه به این سرویس‌ها دسترسی خواهد داشت. در ادامه به بررسی گزینه‌های موجود خواهیم پرداخت.

۲- تعامل مستقیم Clientها با میکروسرویس‌ها:



تعامل مستقیم Clientها و میکروسرویس‌ها

به صورت تئوری می‌توانیم متصور شویم که هر Client در صورت نیاز به صورت مستقیم با Microservice‌ها تعامل خواهد کرد. هر میکروسرویس Endpoint اختصاصی خود را دارد و به راحتی می‌توان با آن تعامل کرد مثل:

Comments.Api.Digikala.ir

■ Suggestion.Api.Digikala.ir

■ Faq.Api.Digikala.ir

با کمی دقت متوجه خواهیم شد که برای اینکه یک صفحه ساده ایجاد شود اپلیکیشن ما نیاز دارد که به چندین Endpoint متفاوت درخواست ارسال کند. متاسفانه هر چند این روش در نگاه اول راحل بدیهی و ساده‌ای به نظر می‌رسد اما مشکلات فراوانی به همراه خواهد داشت. بزرگترین مشکل تعداد زیاد درخواست‌هایی است که باید برای ساخت یک صفحه ارسال شود و با بالارفتن تعداد درخواست کار ساخت و مدیریت صفحه نیز به شدت سخت و پیچیده خواهد شد. در یک اپلیکیشن ساده مانند مثال بالا به سادگی نیاز به ارسال بیش از ۱۵ درخواست برای ساخت صفحه داریم. شرکت آمازون در مطلبی مرتبط توضیح داده بود که چگونه برای نمایش صفحه محصولات خود نیاز به استفاده از قریب به صدها سرویس داشته‌است.

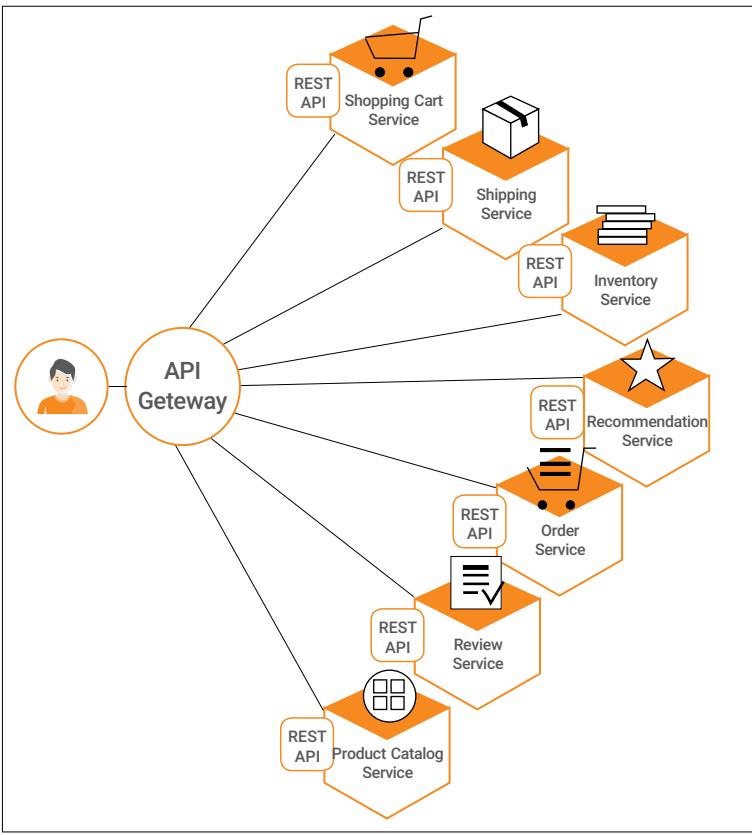
مشکل بعدی مربوط به استفاده از پروتوكلهای متفاوت برای ارائه API می‌شود. با توجه به اینکه هر میکروسرویسی می‌تواند به طور اختصاصی و با توجه به نیازهای خود API‌های خود را ارائه دهد ممکن است یک سرویس از Thrift استفاده و کند و دیگری از AMQP و کاملاً محتمل است که در این بین از پروتوكلهایی استفاده شود که خیلی Web Friendly نیستند.

سومین مشکل در این روش مربوط به Refactor کردن میکروسرویس‌ها می‌شود. در طول زمان ممکن است متوجه اشتباهاتی در طراحی خود بشویم و نیاز داشته باشیم که دو یا چند میکروسرویس را با هم ادغام

کنیم یا برعکس یک سرویس را به چندین سرویس تبدیل کنیم. در حالتی که Client‌ها به طور مستقیم با سرویس‌های ما تعامل کنند احتمالاً این تغییرات تاثیر منفی فراوانی رو Client‌های ما خواهد گذاشت و کار Refactoring بسیار پرخطر و پر ریسک خواهد شد. (به هر حال اگر در چنین شرایطی تصمیم به Refactor گرفتید به طور اکید توصیه می‌کنم تا زمانی که آب‌ها از آسیاب بیوفته یه جایی خودتونو مخفی کنید.) به خاطر تمامی دلایل بالا و بسیاری دلایل دیگری که شاید با کمی بررسی به آن‌ها خواهید رسید استفاده از این روش مناسب نیست و باید به فکر جایگزینی برای این روش باشیم.

۳_ حل مشکلات با استفاده از API Gateway

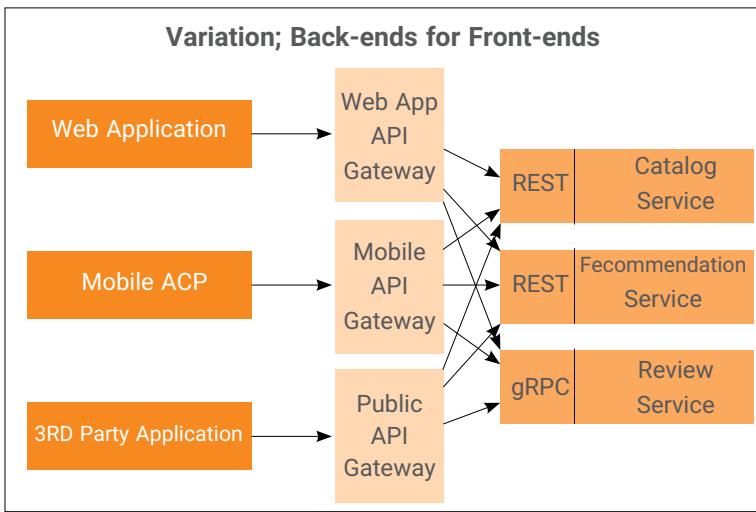
روش بهتری برای تعامل Client‌ها با میکروسرویس‌ها وجود دارد که آن را با نام API Gateway می‌شناسیم. در این روش تنها نقطه ورود برنامه ما یک Gateway است و راه ارتباطی Client‌ها با Microservice‌ها همین است. در صورتی که با الگوی Facade آشنایی داشته باشید API Gateway عملکردی شبیه به این الگو دارد. در این الگو به جای اینکه برای انجام یک کار با چندین API مختلف تعامل کنیم به سادگی با یک API تعامل می‌کنیم و پیچیدگی‌ها و معماری داخلی ما از چشم استفاده کننده پنهان می‌ماند. صدای زدن چندین سرویس مختلف و ترکیب نتیجه و بازگرداندن نتیجه نهایی مواردی است که باید داخل API Gateway ما کپسوله شود و استفاده کننده نهایی به سادگی و با صدای زدن یک API نتیجه دلخواه خود را دریافت کند.



API Gateway

تمامی درخواست‌های کاربران به API Gateway تحویل داده می‌شود و در API Gateway مسیریابی به هر سرویس، تعامل با پروتوكلهای مختلف و ترکیب نتیجه به دست آمده از هر میکروسرویس انجام می‌شود. یکی دیگر از کارهای خوبی که در این زمینه می‌توان انجام داد پیاده‌سازی Gatewaysی تخصصی برای هر Client است. مسلماً تمام داده‌هایی که در صفحه مانیتور نمایش داده می‌شود مناسب نمایش در صفحه یک

گوشی موبایل نیست. پس بهتر است به ازای هر Client یک Gateway داشته باشیم که صرفاً داده‌های آن Client را فراهم کند.



پیاده سازی Gateway تخصصی

۱_۳_ مزایا و معایب استفاده از API Gateway

مثل هر کار دیگری استفاده از API Gateway هم مزایا و معایب خاص خود را دارد که ابتدا به مزایای آن می‌پردازیم. بزرگترین مزیت آن از بین بردن معایب روش دسترسی مستقیم است. عدم وابستگی به معماری داخلی سیستم ما باعث می‌شود کار Refactoring ساده‌تر قابل اجرا باشد و دیگر برای ترکیب یا تجزیه سرویس‌های مختلف دغدغه‌های قبل را نداشته باشیم (پیدا کردن جایی تا زمانی که آبها از آسیاب بیوافتد). ارائه API تخصصی برای هر Client باعث افزایش بهره‌وری و بهبود خروجی‌ها و در یک کلام UX بهتر می‌شود. کاهش تعداد درخواست‌های ارسالی از Client هم مورد بعدی است که بهره‌وری کار را بالاتر می‌برد.

در کنار این مزایا اما چند ایراد نیز می‌توان به استفاده از این روش گرفت. بزرگترین ایراد این روش اضافه شدن یک مازول بزرگ به سیستم است که باید همیشه سرحال و آنلاین باشد و در صورتی که عملکرد درستی ارائه نکند کل سیستم با مشکل مواجه خواهد شد. با توجه به اینکه تعامل با هرکدام از میکروسرویس‌ها باید در API Gate- way پیاده‌سازی شود و به ازای هر Client هم نیاز داریم که پیاده‌سازی API Gateway اختصاصی داشته باشیم این احتمال وجود دارد که همین به سدی برای تیم توسعه تبدیل شود. زمانی که یک سرویس به روز می‌شود Client‌ها باید منتظر بمانند تا این به روزرسانی در Gateway ارائه شود. به همین دلیل باید توسعه API Gateway ما طوری باشد که به سادگی قابل تغییر و به روزرسانی باشد.

با وجود تمامی این مشکلات اما نکات مثبت استفاده از این الگو به قدری زیاد است که نمی‌توان به سادگی از این الگو چشم پوشی کرد.

۳_۲_۱_ پیاده‌سازی API

حال که با مزایا و معایب API Gateway آشنا شدیم به بررسی چند نکته در رابطه با پیاده‌سازی آن خواهیم پرداخت.

۳_۲_۱_ مسئله بهره‌وری و مقیاس‌پذیری:

احتمالاً تعداد انگشت‌شماری شرکت در دنیا مانند Netflix وجود دارند که نیاز دارند روزانه به میلیون‌ها و میلیاردها درخواست پاسخ بدهند و از دسترس خارج شدن آن‌ها حتی برای چند لحظه غیرقابل قبول باشد. با این حال با توجه به شرایطی که بررسی شد، بهره‌وری بالا و

قابلیت مقیاس‌پذیری از نیازهای اولیه هر API Gateway است. ابزارها و زبان‌های مختلفی وجود دارد که می‌تواند این ویژگی‌ها را در اختیار شما قرار دهد که برای مثلاً می‌توان به Node.js و .Net Core اشاره کرد.

۲_۳_۱_ استفاده از مدل برنامه‌نویسی Reactive

در بعضی موارد می‌توان به سادگی درخواست‌های ورودی را به یک مسیر جدید ارسال کرد و نتیجه را دریافت کرد و به کاربر بازگرداند. در بعضی موارد هم ممکن است یک درخواست ورودی به چندین سرویس ارجاع داده شود و در نهایت نتیجه تمامی این درخواست‌ها با هم ترکیب شود و به کاربر بازگردانده شود. در بعضی موارد هم ممکن است یک درخواست نیاز باشد از چند سرویس استفاده کند اما ترتیب و توالی استفاده از این سرویس‌ها اهمیت داشته باشد. برای مثال زمانی که قرار است به کاربری کالا پیشنهاد داده شود ابتدا لازم است تنظیمات و علائق کاربر از سرویس کاربران دریافت شده و سپس با اطلاعات دریافتی کالاهای پیشنهادی پیدا شده و در اختیار کاربر قرار بگیرد.

با توجه به شرایط توضیح داده شده احتمالاً استفاده از روش‌های معمول برنامه‌نویسی خیلی زود ما را وارد جهنمی از کدهای پیچیده و غیرقابل خواندن و تغییر می‌کند. پس بهتر است به روشی توسعه خود را انجام دهیم که مناسب شرایط باشد. در این شرایط به نظر می‌رسد راهکار بهینه‌تری نسبت به روش معمول Reactive Programming برنامه‌نویسی باشد.

۲_۳_۲_ راهکارهای تعامل:

در یک API Gateway نیاز است با سرویس‌های مختلف تعامل

انجام شود و اصطلاحاً Inter-Process Communication سرویس‌های مختلف ممکن است راهکارهای متفاوتی برای تعامل در اختیار ما قرار دهند. ممکن است از روش‌های Async مثل AMQP یا Sync مثل HTTP و Thrift استفاده شود. به هر حال بدون توجه به روش‌های تعامل API Gateway مورد نظر ما باید بتواند با تمامی این روش‌ها ارتباط برقرار کند.

۳_۳_۴_ یافتن آدرس سرویس‌ها:

وقتی Gateway را پیاده‌سازی می‌کنیم باید به این موضوع فکر کنیم که نیاز داریم آدرس سرویس‌های مختلف را پیدا کنیم. در روش‌های قدیمی احتمالاً نرم‌افزارهای ما به راحتی روی یک سرور نصب می‌شوند و دانستن آدرس آن‌ها کار سختی نخواهد بود. اما در روش‌های جدید و استفاده از سرویس‌های ابری آدرس دیگر به سادگی و ثابت به دست نمی‌آید پس باید قبل از هر مسئله‌ای به یافتن آدرس میکروسرویس‌ها بیاندیشیم. در قسمت‌های بعد به طور مفصل راجع به این مشکل و راه حل آن صحبت خواهیم کرد.

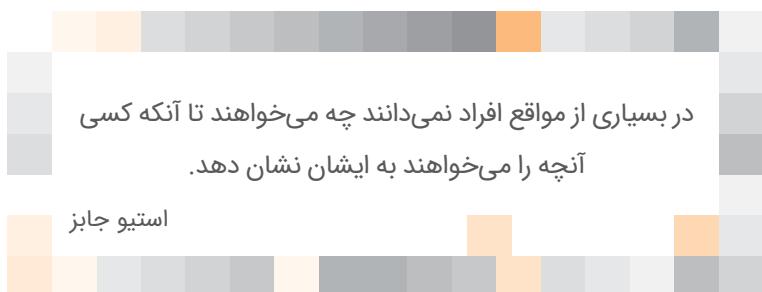
۳_۳_۵_ مدیریت خطاهای:

مشکل دیگری که هنگام توسعه یک API Gateway باید به آن بیاندیشیم Partial Failure است. یعنی زمانی که یک درخواست کلی می‌آید و بخشی از درخواست قابل پاسخ‌گویی نیست. مثلاً در سیستم فروشگاه و صفحه جزئیات فروشگاه یکی از سرویس‌ها در دسترس نباشد. مثلاً سرویس پیشنهاد کالا در دسترس نباشد. API Gateway باید این قابلیت را داشته باشد که در این شرایط به جای اینکه کل

درخواست را لغو کند، داده‌های بخش‌های صحیح را به دست آورد و برای بخش‌های مشکل دار خطای مدیریت کند. برای مثال داده‌های کش شده داشته باشد که در این شرایط جایگزین داده‌های آنلاین شود. یا مثلاً در صورتی که سرویس پیشنهاد کالا قطع باشد ۱۰ کالای پرفروش را بازگرداند.

۴- جمع‌بندی:

در این قسمت راجع به یکی از الگوهای پرکاربرد و شرایط و نیازمندی‌های توسعه آن صحبت کردیم. پیاده‌سازی یک API Gateway خالی از لطف نیست و می‌تواند به درک بهتر چگونگی پیاده‌سازی این الگو کمک کند. با این حال برای پروژه‌های عملیاتی با توجه به شرایط و نیازهای پروژه استفاده از ابزارهای آماده برای این کار مانند Kong، Aws API Gateway یا Ocelot گزینه‌های بهتری است.



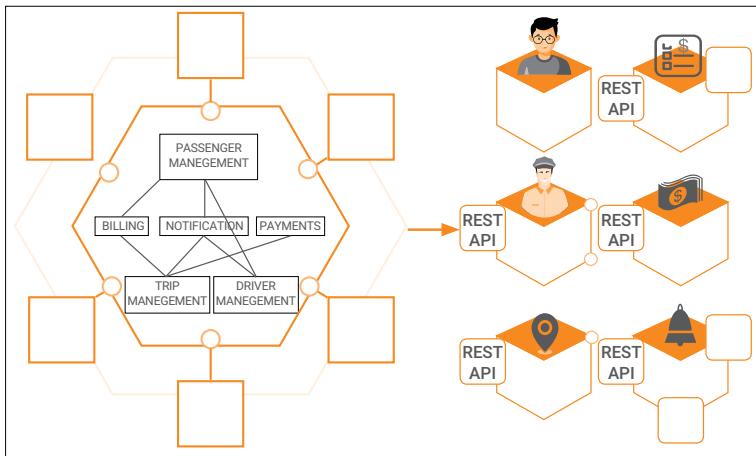
فصل سوم: ارتباط بین سرویس‌ها



- انواع روش‌های تعامل
- تعریف API‌ها
- تکامل API‌ها
- مدیریت بحران هنگامی که بخشی از سیستم دچار مشکل می‌شود.

مقدمه:

در برنامه‌ها Monolithic با توجه به اینکه کل برنامه ما در یک نرم‌افزار با یک زبان برنامه‌نویسی پیاده‌سازی شده است ارتباط بین بخش‌های مختلف به سادگی صدا زدن یک تابع انجام می‌شود. اما در میکروسرویس‌ها با توجه به اینکه بخش‌های مختلف برنامه ما روی چندین سیستم مختلف به صورت توزیع شده اجرا می‌شوند امکان استفاده از روش قبل وجود ندارد. هر بخش از نرم‌افزار ما به صورت یک پروسه کاملاً جداگانه و ایزوله اجرا می‌شود. در میکروسرویس‌ها ارتباط بین بخش‌های مختلف اصطلاحاً از روشی به نام Inter process Communication که از این به بعد به اختصار IPC می‌گوییم انجام می‌شود. پیش از آنکه به سراغ انواع روش‌ها و تکنولوژی‌های پیاده‌سازی IPC برویم بباید نگاهی به موارد مختلفی که هنگام طراحی ارتباط بین سرویس‌ها باید مد نظر داشته باشیم بیاندازیم.



أنواع روش‌های تعامل:

پیش از طراحی و پیاده‌سازی IPC بهتر است در مورد چگونگی برقراری ارتباط بین سرویس‌ها کمی تأمل کنیم. انواع مختلفی از روش‌های مختلف تعامل بین سرویس‌ها و کلاینت‌ها وجود دارد. چگونگی برقراری ارتباط بین سرویس‌ها مختلف از دو جنبه مختلف و کلی قابل بررسی است. جنبه اول اینکه ارتباط بین کلاینت و سرویس‌دهنده یک به یک است یا یک به چند:

■ یک به یک: هر درخواست کلاینت دقیقاً توسط یک سرویس دریافت و پردازش می‌شود.

■ یک به چند: هر درخواست کلاینت توسط چند سرویس دریافت و پردازش می‌شود.

جنبه بعدی قابل بررسی در ارتباط بین سرویس‌ها Sync یا Async بعدن ارتباط است.

■ ارتباط Sync: کلاینت در یک بازه زمانی خاص توقع دریافت پاسخ از سرویس‌دهنده را دارد و معمولاً در این بازه زمانی معطل می‌ماند.

■ **ارتباط Async:** تفاوتی نمی‌کند که پردازش درخواست پاسخی در برخواهد داشت یا خیر. در هر صورت کلاینت در انتظار پاسخ درخواست نخواهد ماند و به کار خود ادامه خواهد داد و پاسخ بعداً به صورت داده خواهد شد.

در جدول زیر انواع روش‌های ارتباطی با توجه به جنبه‌های بالا را مشاهده می‌کنید:

یک به چند	یک به یک
-	Request/ Response
Publish/ Subscrib	Notification
Publish/ Async Response	Request/ Async Response

در ادامه به معرفی هر یک از این روش‌های ارتباطی خواهیم پرداخت.

ابتدا به سراغ روش‌های یک به یک می‌رویم که این روش‌ها عبارتند از:

■ **روش Request/Response:** احتمالاً یکی از ساده‌ترین و مرسوم‌ترین روش‌های ارتباطی همین روش است. کلاینت درخواستی را به سرویس دهنده ارسال می‌کند و منتظر می‌ماند تا پاسخ مناسب از سمت سرویس دهنده ارسال شود. در این روش کلاینت در انتظار پاسخ خواهد ماند و با توجه به اینکه در این بازه زمانی احتمالاً Thread Block شده است در صورت طولانی شدن انتظار سیستم کلاینت دچار اختلال خواهد شد.

■ **روش دوم Request یا Notification یک طرفه:** کلاینت درخواست یا پیامی را برای یک سرویس خاص ارسال می‌کند اما انتظار پاسخ خاصی را ندارد و یا اینکه اصلاً پاسخی وجود ندارد.

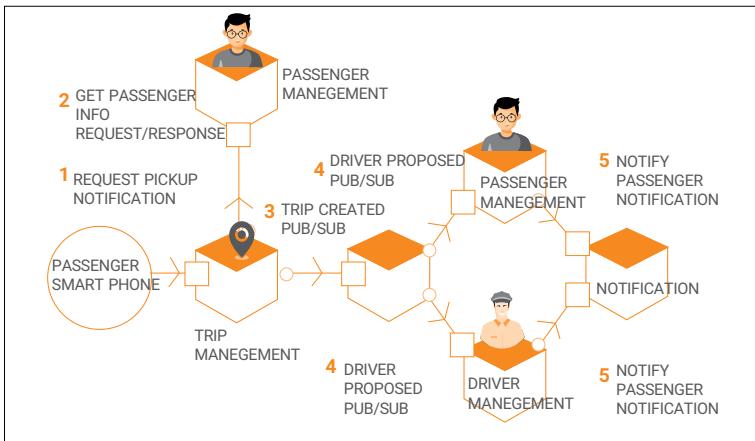
■ **روش Request/Async Response:** کلاینت درخواستی را برای سرویس ارسال می‌کند و با اینکه توقع پاسخ دارد اما در انتظار دریافت پاسخ نمی‌ماند. در این روش ارتباط به گونه‌ای طراحی و پیاده‌سازی می‌شود که کلاینت نیازی به پاسخ فوری ندارد و می‌داند دریافت پاسخ ممکن است با تأخیر همراه باشد.

روش‌های ارتباطی یک به چند نیز به گروههای زیر تقسیم‌بندی می‌شوند:

■ **روش Publisher/Subscribe:** کلاینت یک پیام در سیستم ارسال می‌کند و با توجه به شرایط صفر تا هر تعداد دیگری سرویس‌دهنده انتظار این پیام را می‌کشند و با دریافت این پیام شروع به پردازش اختصاصی پیام دریافتی می‌کنند.

■ **روش Publish/Async Responses:** پیامی از طرف کلاینت ارسال می‌شود و توقع این وجود دارد که پاسخی از طرف برخی سرویس‌ها برای این پیام ارسال شود.

در پیاده‌سازی میکروسرویس‌ها از این روش‌های ارتباطی استفاده می‌شود. یک سرویس با توجه به شرایط طراحی و نیازمندی خاص خود ممکن است از یکی از این روش‌ها صرفاً استفاده کند و سرویس دیگر ممکن است از ترکیبی از این سرویس‌ها استفاده نماید.



همانطور که در تصویر مشاهده می‌کنید سرویس‌های مختلف از انواع روش‌های ارتباطی برای انجام ماموریت کاری خود و رسیدن به هدف نهایی که ثبت درخواست تاکسی برای یک مسافر است استفاده می‌کنند. از نرم‌افزار موبایل یک سرویس Trip Management Notification برای سرویس ارسال Request/Response Trip Management با روش Trip Management می‌شود. سرویس Trip Management با روشنگاری مطلع می‌شود. سپس سرویس‌ها اطلاع پذیری حساب کاربر مطلع می‌شود. این سرویس‌ها اطلاعات Trip ایجاد می‌کند و با روشنگاری Pub/Sub به بقیه سرویس‌ها اطلاع می‌دهد. ادامه درخواست‌ها تا زمان نهایی شدن درخواست را در تصویر می‌توانید مشاهده کنید.

حال که با انواع روش‌های ارتباطی بین سرویس‌ها آشنا شدیم، باید با هم نحوه تعریف API‌ها و نکاتی که باید هنگام طراحی آن‌ها مد نظر داشته باشیم را بررسی کنیم.

تعریف API‌ها:

هنگامی که نیاز به برقراری ارتباط بین سرویس‌ها و کلاینت‌ها داریم

باید API‌هایی تعریف کنیم و قراردادی برای این API‌ها بین کلاینت و سرویس برقرار باشد. بدون توجه به روش IPC که برای بخش‌های مختلف سیستم انتخاب می‌کنید تعریف یک ساختار خوب و دقیق از هر API می‌تواند موفقیت یک سرویس را تضمین کند. بحث‌های زیادی در مورد این مسئله صورت گرفته است که حتی بهتر است هنگام طراحی یک سرویس از روش API First استفاده کنیم و ابتدا API‌های سرویس را تعریف کنیم و با توجه به API‌های ارائه شده اقدام به طراحی و پیاده‌سازی خود سرویس کنیم. طرفداران این روش طراحی به این نکته قائل هستند که طراحی API‌ها در ابتدا شанс تولید سرویسی که خدمات درستی به کلاینت‌های خود می‌دهد را افزایش می‌دهد. در ادامه خواهیم دید که انتخاب روش IPC چگونه بر روای طراحی و پیاده‌سازی API‌ها تاثیر خواهد گذاشت.

تکامل API‌ها:

در گذر زمان API‌های ارائه شده توسط یک سرویس به دلایل مختلف تغییر و تکامل خواهد یافت. در یک برنامه Monolithic API یک قسمت بسیار ساده‌است و با توجه به اینکه استفاده کننده‌های یک API با خطأ مواجه خواهند شد تقریباً می‌توان مطمئن بود با تغییر API تمامی کلاینت‌های آن نیز به روز خواهند شد. اما در میکروسرویس‌ها اطمینان از به روز بودن تمامی کلاینت‌های کاری بسیار دشوار و در بعضی موارد غیرممکن است. در اغلب موارد امکان اجبار کلاینت‌ها به بروزرسانی وجود ندارد. بعضاً این احتمال وجود دارد که API‌های شما

تغییر نکند و صرفاً تکامل بباید و در این شرایط می‌توان توقع داشت که شما کلاینت‌هایی داشته باشید که با نسخه‌ها و امکانات قدیمی کار می‌کنند یا کلاینت‌هایی که خود را به روز کرده و از ویژگی‌های جدید API‌های شما استفاده می‌کنند. در هر صورت چه شما تغییر داشته باشید و چه تکامل داشتن استراتژی‌هایی برای برخورد با شرایط مختلف و کلاینت‌های مختلف از نیازهای اولیه تیم توسعه است.

چگونه شما می‌توانید تغییرات را در نسخه‌های مختلف API خود مدیریت کنید؟ برخی تغییرات جزئی است و قابل مدیریت و به سادگی می‌توان تغییرات را به گونه‌ای انجام داد که Backward Compatible باشد. مثالی از این تغییرات می‌تواند اضافه شدن یک خاصیت به پاسخی باشد که سرویس برای یک درخواست خاص ارسال می‌کند، در این شرایط تغییر به شکلی است که کلاینت به عملکرد قبلی خود بدون تغییر می‌تواند ادامه دهد و فقط از ویژگی جدید بهره‌مند نمی‌شود. هنگام طراحی سرویس‌ها و کلاینت‌ها توجه به اصل Robustness می‌تواند مفید به فایده باشد. کلاینت‌هایی که از نسخه‌های قدیمی یک API استفاده می‌کنند باید بتوانند به کار خود ادامه دهند. سرویس‌ها باید برای ویژگی‌های اضافه‌ای که به ساختار درخواست اضافه می‌کنند مقدار پیش‌فرض تخصیص دهند و کلاینت‌ها هم باید توانایی صرف‌نظر کردن از ویژگی‌هایی که برای آن‌ها تعریف نشده‌است را داشته باشند. انتخاب روش ارتباطی که به شما به سادگی قابلیت ارتقا و تغییر API‌های سرویس‌ها را بدهد بسیار مهم و حیاتی است.

با همه این تفاسیر در برخی موارد ممکن است شما نیاز به تغییراتی داشته باشید که قابلیت Backward Compatibility را ندارند و به هر حال

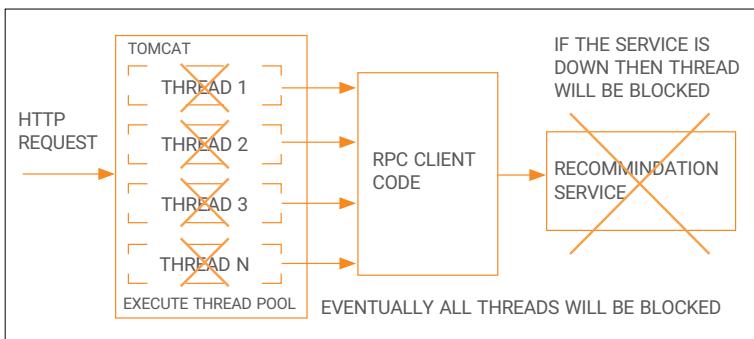
کلاینت‌ها از کارخواهند افتاد. در این شرایط پشتیبانی از API‌های قدیمی تا زمان اطمینان از ارتقا پیدا کردن همه کلاینت‌ها تنها راه ممکن برای داشتن یک سرویس قدرتمند و قابل اطمینان است. (البته می‌توانید خیلی ساده نسخه‌های قدیم را از دسترس خارج کنید. مطمئن باشید کلاینت‌های بی‌دفاع چاره‌ای جز ارتقا سریع خود نخواهد داشت. همینقدر بی‌فکر و از خود راضی). برای مثال اگر از REST API‌ها استفاده می‌کنید می‌توانید نسخه API خود را در URL دخیل کنید تا کلاینت‌ها بتوانند با نسخه مناسب خود ارتباط برقرار کنند.

مدیریت بحران هنگامی که بخشی از سیستم

دچار مشکل می‌شود:

همانطور که در قسمت دوم هم گفته شد، با توجه به اینکه سرویس‌دهنده‌ها و کلاینت‌ها کاملاً مجزا هستند این احتمال وجود دارد که در زمانی که یک کلاینت درخواستی را برای سرویسی ارسال می‌کند سرویس امکان دریافت پیام و ارسال پاسخ مناسب را نداشته باشد. ممکن است در زمان ارسال درخواست سرویس‌دهنده به منظور ارتقا و به روزرسانی از دسترس خارج باشد یا به خاطر فشار بسیار بالا سرعت ارائه خدمات پایین‌تر از حد انتظار کلاینت‌ها باشد. اگر به سراغ مثال فروشگاه در مطالب قبلی بازگردیم فرض کنید سرویس پیشنهاد کالا در زمان نیاز به این سرویس از دسترس خارج باشد. یک پیاده‌سازی ناصحیح و فکر نشده ممکن است به شکلی باشد که کلاینت برای مدت زمان طولانی در انتظار پاسخ از طرف سرویس پیشنهاد کالا بماند. این روش طراحی و پیاده‌سازی

نه تنها UX مناسبی ندارد بلکه ممکن است موجب هدر رفتن منابع موجود در سیستم نیز بشود.



برای جلوگیری از بروز چنین مشکلاتی طراحی سیستم به شکلی که قابلیت مدیریت چنین خطاهایی را داشته باشد بسیار حیاتی است. در صورتی که سیستم با بروز خطا در قسمتی از آن توانایی ادامه حیات داشته باشد اصطلاحاً می‌گوییم سیستم تحمل خطا را دارد یا -Tolerance است. توجه به این نکته که تحمل خطا در سیستم‌های توزیع شده یک نیازمندی است نه یک ویژگی بسیار مهم است. هنگام مواجه شدن با خطا در یکی از قسمت‌های نرم‌افزار استراتژی‌های متفاوتی را می‌توانیم در پیش بگیریم که در ادامه به بررسی این استراتژی‌ها خواهیم پرداخت.

در نظر گرفتن زمان Timeout: هنگامی که از روش‌های Sync در توسعه نرم‌افزار استفاده می‌کنید از انتظار بی‌پایان برای دریافت پاسخ به شدت دوری کنید و با توجه به شرایط سرویس‌ها و معماری سیستم مدت زمانی را به عنوان Timeout در سیستم تنظیم کنید که در صورتی که در

زمان مناسب پاسخی ارسال نشد کلاینت عملیات مدیریت خط را شروع کرده و از تله انتظار بیپایان رها شود.

محدود کردن تعداد درخواست‌های بدون پاسخ: در صورتی که تعداد زیادی درخواست را برای یک سرویس ارسال کرده و هیچ پاسخی دریافت نکنیم با ارسال درخواست جدید احتمالاً فقط صف درخواست‌های بدون پاسخ طولانی‌تر شده و نتیجه‌ای جز هزینه بالاتر هنگام وقوع خط را برخواهد داشت. پس بهتر است تعداد درخواست‌های در انتظار پاسخ را محدود کنیم و در شرایطی که به حداقل تعداد درخواست‌های قابل قبول رسیدیم درخواست‌های جدید را بدون فوت وقت و درج در صف درخواست‌ها لغو کنیم.

استفاده از الگوی Circuit Breaker: هنگامی که تعداد زیادی از درخواست‌ها به یک سرویس خاص با خط مواجه می‌شود می‌توان حدس زد که سرویس مقصد دچار مشکل شده است. در این شرایط ارسال درخواست برای این سرویس به جز هدر دادن منابع و انتشار خط را سیستم دستاورد دیگری ندارد. برای این مشکل‌ها خوب است از روشی استفاده کنیم که مهندسین برق استفاده می‌کنند و مازولی مانند فیوز در سیستم قراردهیم. در زمان بروز خط همانطور که فیوز قطع می‌شود مازولی ما نیز سرویس را برای مدت‌زمانی خاص از دسترس خارج کرده و تمامی درخواست‌ها به این سرویس فوراً Reject شوند. بعد از گذشتن بازه زمانی خاص Circuit Breaker اجازه عبور تعداد کمی درخواست را خواهد داد. در صورتی که این درخواست‌ها با موفقیت پاسخ داده شوند سیستم مطمئن

می‌شود مشکل حل شده و به حیات خود ادامه می‌دهد و در صورت بروز مشکل در پاسخ به این تعداد اندک درخواست احتمال ادامه دار بودن خطای وجود دارد و مجدداً سیستم برای مدت زمانی خاص از دسترس خارج بوده و تمامی درخواست‌ها سریعاً Reject می‌شود.

در نظر گرفتن Fallback: برای شرایط بحرانی و بروز مشکل راهکارهای جایگزین در نظر بگیرید. برای مثلاً در صورتی که سرویس پیشنهاد کالا قادر به ارائه خدمات نبود می‌توان ۱۰ کالای هم خانواده کالای منتخب یا ۱۰ کالای جدید اضافه شده به سیستم را باز گرداند. یا به عنوان یک راهکار دیگر می‌توان نتیجه هر درخواست را کش کرد و در موقع لزوم از اطلاعات کش شده برای پاسخ به کاربر استفاده نمود.

برای پیاده‌سازی این ویژگی‌ها می‌توانید از Netflix Hystrix استفاده کنید و با تنظیماتی که در این سیستم وجود دارد می‌توانید به بسیاری از الگوهایی که در بالا توضیح داده شد دست پیدا کنید.

جمع‌بندی:

در این فصل به بررسی روش‌های ارتباط بین سرویس‌ها پرداختیم. تمامی سیستم‌های توزیع شده نیاز به برقراری ارتباط با سایر سرویس‌ها دارند و با توجه به نیازمندی پروزه انواع روش‌های متفاوت ارتباط وجود دارد که در این فصل بررسی شد. در ادامه نیز به بررسی بایدها و نبایدهای طراحی API‌ها و ارائه راهکارهای ارتباطی پرداختیم. در فصل بعد تکنولوژی‌ها و پروتکل‌های ارتباطی را بررسی خواهیم کرد.

فصل چهارم: تکنولوژی‌های ارتباطی



- ارسال پیام به صورت Async
- ارتباط Request/Response و Sync
- سرویس‌های REST
- مزایای HTTP
- معایب HTTP
- ساختار پیام‌ها

مقدمه:

وقتی در مورد ارتباط صحبت می‌کنیم دو گروه از عناصر اصلی قابل شناسایی هستند. شرکت‌کنندگان در یک رابطه و پیام‌های ارتباطی دو عنصر اصلی هر ارتباطی هستند که در این فصل قصد داریم در مورد این دو مورد صحبت کنیم.

در یک ارتباط IPC تکنولوژی‌های مختلفی قابل انتخاب است. برای مثال برای یک ارتباط از نوع Request/Response و به روش Sync می‌توان API Rest هایی بر اساس پروتکل HTTP توسعه داد یا برای توسعه سرویس‌های خود از Thrift استفاده کنید. در مقابل اگر نیاز به برقراری ارتباط Async داشته باشیم می‌توانیم به سراغ STOMP یا AMQP برویم.

برای فرمت پیام‌های ارسالی و دریافتی نیز گزینه‌های زیادی وجود دارد. یکی از معمول‌ترین انتخاب‌ها قالب‌های محیوب JSON و XML است که قابلیت خوانایی بالایی نیز دارد. اگر خوانایی برای انسان اهمیت نداشته باشد یا نیاز به بهره‌وری بالای داشته باشیم Avro یا Protocol Buffer انتخاب‌های مناسبی به نظر می‌رسند.

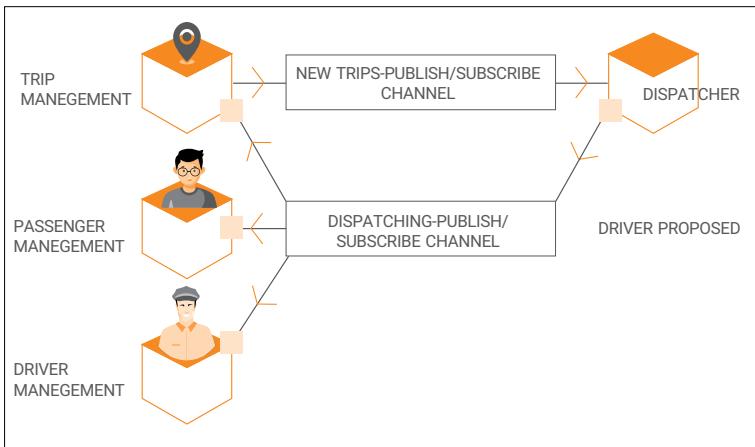
ارسال پیام به صورت Async

هنگامی که از روال‌های Messaging استفاده می‌کنیم یک ارتباط Async برقرار خواهیم کرد. کلاینت یک پیام را برای سرویس دهنده ارسال می‌کند. در صورتی که نیاز باشد برای پیام دریافت شده پاسخی ارسال شود، این پاسخ در قالب پیامی جداگانه از طرف سرویس دهنده برای سرویس گیرنده ارسال می‌گردد. از آنجایی که این یک ارتباط Async است لزومی به منتظر ماندن کلاینت برای دریافت پاسخ وجود ندارد، در عوض هنگام توسعه کلاینت به گونه‌ای توسعه داده می‌شود که عدم دریافت پاسخ فوری از طرف سرویس دهنده اختلالی در عملکرد کلی کلاینت ایجاد نکند. هر Message در این روش ارسال پیام شامل یک Header است که متادیتاهای مرتبط با پیام را نگهداری می‌کند و هدف اصلی در بدن پیام جایگذاری می‌شود. در این روش پیام‌ها از طریق Chanel‌ها منتقل می‌شوند. ارسال کننده‌های مختلف این امکان را دارند که پیام‌های خود را از طریق یک کانال ارتباطی خاص جابجا کنند. در مقابل دریافت کننده‌های متفاوتی نیز این امکان را دارند که پیام‌های یک کانال را دریافت کنند. به طور کلی دو نوع کانال ارتباطی Point-to-Point و Publish-Subscribe قابل تصور است.

ارتباط Point-to-Point: نوعی ارتباط یک به یک است که ارسال کننده پیام، هدف خاصی برای ارسال پیام دارد و پیام به یک دریافت کننده خاص می‌رسد.

ارتباط Publish-Subscribe: در این روش کانال ارتباطی پیام را از ارسال کننده دریافت می‌کند و این احتمال وجود دارد که به صفر تا بی‌نهایت

دریافت کننده پیام را برساند. یکی از راههای ارتباطی One-To-Many استفاده از این روش است. در تصویر زیر استفاده از کanal ارتباطی Publish-Subscribe را مشاهده می‌نمایید.



هنگامی که یک سفر جدید ایجاد می‌شود، سرویس Trip Management یک پیام ایجاد شدن سفر را ارسال می‌کند و هر سرویسی که علاقه‌مند به اطلاع از این واقعه در سیستم باشد، می‌تواند در کanal ارتباطی ثبت‌نام کند. برای مثال در سیستم بالا سرویس‌های Dispatch-er در مورد ایجاد یک سفر جدید حساسیت دارد و بعد از وقوع این واقعه در سیستم عملیات خاصی را شروع می‌کند. سیستم‌های مختلفی برای انتخاب به عنوان زیرساخت ارسال و دریافت پیام وجود دارد که بهتر است سیستمی را انتخاب کنید که از زبان‌های برنامه‌نویسی مختلف پشتیبانی کند.

برخی از سیستم‌های موجود پروتکل‌های استانداردی را برای انتقال پیام انتخاب و استفاده کرده اند و در مقابل سیستم‌هایی هم وجود دارند که

روش‌های ابداعی و اختصاصی خود را برای انتقال پیام پیاده‌سازی کرده‌اند. تعداد زیادی سیستم Open Source برای انتخاب وجود دارد مثل Rab-NSQ، bitMQ، Apache Kafka، Apache ActiveMQ قابل تشخیص و بررسی بین این سیستم‌ها وجود دارد اما تمامی این سیستم‌ها از ساختارهایی برای ارسال پیام و ایجاد و مدیریت کانال پشتیبانی می‌کنند و تلاش می‌کنند ویژگی‌هایی مثل قابلیت اطمینان، بهره‌وری بالا و توزیع شدگی را داشته باشند.

مزایای بسیاری را می‌توان برای استفاده از سیستم‌های Messaging برشمرد که در ادامه برخی از این مزایا را با هم بررسی می‌کنیم.

جداسازی کامل کلاینت از سرویس: در این روش کلاینت پیامی را با فرمت خاص برای یک کانال ارتباطی ارسال می‌کند. در این روش کلاینت کاملاً از روش ارسال پیام و سرویسی که پیام را دریافت می‌کند بی‌اطلاع است. نکته قابل توجه در این روش ارتباطی عدم نیاز کلاینت به دانستن موقعیت سرویس است و نیازی به روال‌های Discovery ندارد.

عدم نیاز به صحت همزمان سرویس و کلاینت: هنگامی که از روشنی مبتنی بر درخواست و پاسخ استفاده می‌کنید مثل HTTP در لحظه برقراری ارتباط کلاینت و سرور هردو باید همزمان اجرا باشند و به درستی خدمات خود را ارائه دهند. اما در این روش این احتمال وجود دارد هنگام ارسال پیام از طرف کلاینت، سرویس قابل دسترسی نباشد و پیام در سیستم باقی می‌ماند تا زمانی که سرویس خدمات رسانی خود را شروع کند و پیام به سرویس تحويل داده می‌شود.

مشهود بودن ویژگی‌های IPC: هنگامی که از IPC استفاده می‌کنیم، در حال استفاده از سرویسی خارج از برنامه جاری هستیم. برخی از روش‌های برقراری ارتباط به گونه‌ای طراحی شده‌اند که این حس جدا بودن به برنامه‌نویس منتقل نشده و با بت همین موضوع هم مواردی مانند احتمال خرابی شبکه یا ... از چشم برنامه نویس پنهان می‌ماند. اما ماهیت برقراری ارتباط به روش Messaging کاملاً این جدا بودن را نمایان می‌سازد. با تمام مزایایی که می‌توان برای این روش ارتباطی قائل شد معايیت نیز وجود دارد که هنگام انتخاب روش برقراری ارتباط باید به آن دقت کرد.

پیچیدگی‌های عملیاتی و زیرساختی: هرچند این روزها با پیشرفت‌هایی که سیستم‌های نرم‌افزاری داشته‌اند کار نصب و راهاندازی آن‌ها به شدت ساده شده‌است و با وجود ابزارهایی مثل داکر این پیچیدگی به صفر میل می‌کند، اما کماکان برای استفاده از این روش نیاز به افزودن عضو جدیدی به سیستم داریم که جدای از مسائل نصب و راهاندازی نیاز به نگهداری و مانیتورینگ دارد.

پیچیدگی‌های پیاده‌سازی: هنگامی که از این روش ارتباطی استفاده می‌کنیم در صورتی که نیاز داشته باشیم به ازای یک درخواست پاسخی نیز دریافت کنیم باید کانالی برای دریافت پاسخ تخصیص دهیم و شناسه‌ای باید به درخواست تخصیص دهیم تا هنگام دریافت پاسخ بتوانیم پاسخ دریافتی را به یک درخواست متصل کنیم. حال که با عملکرد کلی سیستم‌های Messaging آشنا شدیم به سراغ یکی دیگر از روش‌های ارتباطی یعنی Request/Response می‌رویم.

ارتباط Request/Response و Sync

هنگام استفاده از ارتباط متقارن و روش Request/Response کلاینت درخواستی را برای سرویس ارسال نموده و منتظر پاسخ می‌ماند، سرویس دهنده نیز با دریافت یک درخواست شروع به پردازش آن نموده و نتیجه نهایی را به درخواست کننده باز می‌گرداند. تفاوت اصلی بین این روش با روش قبلی در این است که کلاینت فرض می‌کند که سرویس در بازه زمانی معقولی پاسخی به وی خواهد داد.

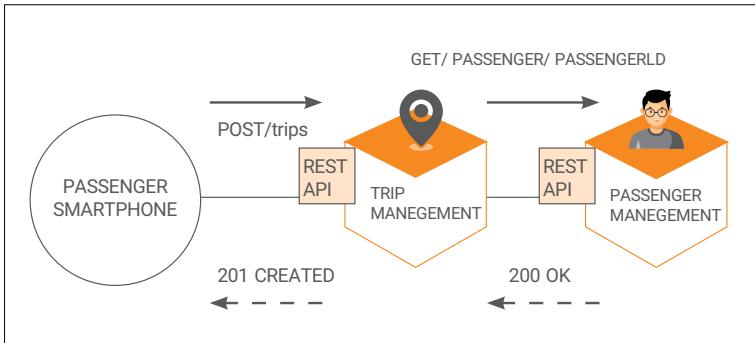
پروتکل‌های زیادی برای پیاده‌سازی این روش ارتباطی وجود دارند که دو تا از مشهور ترین این پروتکل‌ها Rest و Thrift است.

سرویس‌های REST

یکی از متدائل ترین روش‌های پیاده‌سازی IPC که این روزها بسیار مورد توجه توسعه دهنده‌گان قرارداد REST است که با توجه به تعریف‌های داخلی آن معمولاً از پروتکل HTTP استفاده می‌کند.

سرویس‌های REST اساساً بر مبنای Resource‌ها طراحی و پیاده‌سازی می‌شوند که هر Resource یک شی خاص در کسب و کار است، مثل مشتری، محصولات و ... این روش ارتباطی از ترکیب Verb‌ها و آدرس برای دستیابی صحیح به یک Resource استفاده می‌کند. برای مثال دسترسی به آدرس: nikamooz.com/api/course: با توجه به نوع درخواست که می‌تواند GET یا POST باشد قابلیت دریافت یک مجموعه آموزشی یا به روزرسانی آن را انجام می‌دهد. در این روش ارتباطی این امکان وجود دارد که استفاده کننده با توجه به

شرایط، درخواست دریافت نتیجه با فرمتی خاص مثل XML یا JSON را داشته باشد.



بسیاری از برنامه‌نویسان ادعا می‌کنند که بر مبنای پروتکل HTTP توسعه داده‌اند REST Full است، باید توجه داشت که صرفاً ادعای REST Full نبودن نتیجه دلخواه را نمی‌دهد و باید یکسری اصول را نیز REST APIs Must be Hy-pertext-Driven رعایت کنید. برای کسب اطلاعات بیشتر لطفاً REST APIs Must be Hy-pertext-Driven را در گوگل جستجو کنید و مقاله مربوطه را مطالعه کنید. در صورتی که تمایل دارید بدانید چگونه می‌توانید API‌هایی بالغ داشته باشید پیشنهاد می‌کنم مدل بلوغ ریچاردسون نوشته آقای مارتین فاولر را مطالعه کنید.

مثل هر روش و ابزار دیگری استفاده از HTTP به عنوان بستر ارتباطی مزایا و معایبی دارد که در ادامه به بررسی مزایا و معایب این روش می‌پردازیم.

مزایای HTTP:

■ بسیار ساده و آشنا

- سادگی در تست و عملیاتی سازی و وجود ابزارهای بسیار زیاد برای تست مثل PostMan
- به صورت توکار از روش Request/Response پشتیبانی می‌کند.
- همخوانی بسیار مناسب با بسترهای سخت افزاری و امنیتی مثل فایروال‌ها
- عدم نیاز به واسط ارتباطی

معایب HTTP:

- به صورت توکار فقط از Request/Response پشتیبانی می‌کند حتی اگر Response نیاز نداشته باشیم به عنوان جزئی لاینفک از پروتکل پاسخی نیز ارسال می‌شود.
- با توجه به ماهیت ارتباط مستقیم کلاینت و سرویس، هر دو باید همزمان قابلیت ارسال و دریافت پیام را داشته باشند و در صورت عدم دسترسی به یکی از این دو سر ارتباط درچار اختلال خواهیم شد.
- کلاینت‌ها نیاز دارند تا آدرس دقیق سرویس‌ها را بدانند.

ساختار پیام‌ها:

بعد از بررسی راهکارهای ارتباطی نوبت به بررسی خود پیام‌ها می‌رسد. در بین روش‌های مختلف ارتباطی REST و Messaging این قابلیت را به برنامه‌نویس می‌دهند که از بین فرمتهای مختلف مورد دلخواه خود را انتخاب کنند اما برخی روش‌ها مثل Thrift این قابلیت را ندارند. پس در مواردی که نیاز به فرمتهای مختلف داریم قاعده استفاده از روش‌هایی مثل REST بسیار خوب و پذیرفته است.

یکی از ساختارهای کلی ارسال پیام استفاده از ساختار متنی است. فرمتهایی مثل JSON و XML در توسعه نرمافزار بسیار کاربرد دارند. نکته منفی استفاده از این دو روش، خوانایی بالای پیام‌های جابجا شده به این روش است. جدای از قابلیت خوانایی بالا، بهره‌وری پایینی نیز برای استفاده از این روش وجود دارد.

در طرف مقابل می‌توانید از فرمتهایی که به صورت باینری کار تبادل داده‌ها را انجام می‌دهند استفاده نمایید.

جمع‌بندی:

در این فصل سعی کردیم کمی در مورد تکنولوژی‌های موجود برای پیاده‌سازی ارتباطات بین سرویس‌ها صحبت کنیم.

فصل پنجم: آشنایی با Service Discovery

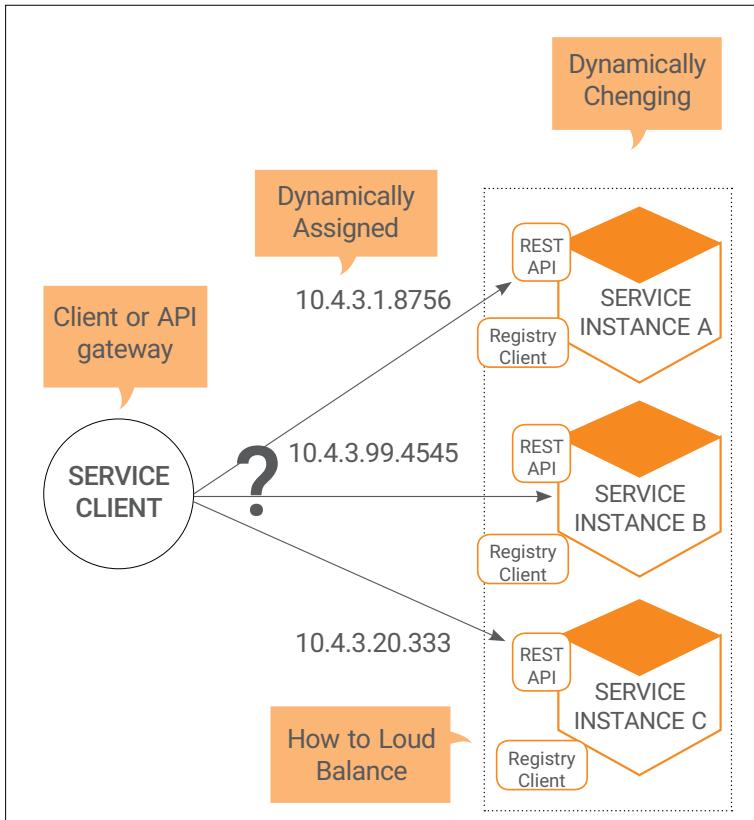


- آشنایی با Service Registry
- آشنایی با Client-Side Discovery
- آشنایی با Server-Side Discovery
- بررسی دقیق Service Registry
- انواع روش‌های مدیریت سرویس‌ها

مقدمه:

بیایید فرض کنیم که در حال توسعه یک نرمافزار هستیم که نیاز دارد یک API را صدا بزند. فارغ از اینکه یک REST API را مورد استفاده قرار می‌دهیم یا Thrift API یا API موردنظر را بدانیم. تا چند سال پیش که نرمافزارها معمولاً در یک سرور نصب می‌شدند آگاهی از آدرس API‌ها کار مشکلی نبوده و معمولاً آدرسی ثابت بود. در چنین شرایطی نهایت آینده نگری که نیاز بود داشته باشیم، قراردادن آدرس API در فایل Config بود که در صورتی که در آینده نیاز به تغییر آدرس داشته باشیم، بتوانیم بدون کامپایل مجدد سورس کد، آدرس را تغییر دهیم.

اما این شرایط در نرمافزارهای مدرنی که روی Cloud اجرا می‌شوند و آدرس آن‌ها دائم تغییر می‌کند و در میکروسرویس‌هایی که دائم *In-Instance* متفاوتی از آن‌ها در آدرس‌های متفاوتی اجرا می‌شود صادق نخواهد بود. یک سرویس ممکن است با توجه به فشار زیادی که روی آن وجود دارد به صورت اتوماتیک Scale-Out شود و در چند آدرس جدید اجرا شود. ممکن است یک سرویس که در یک آدرس وجود دارد به دلیل خطا از دسترس خارج شده و دیگر در آدرس قبلی سرویسی برای ارائه خدمات وجود نداشته باشد.



با توجه به همه این شرایط می‌توانیم نتیجه بگیریم که دیگر آدرس‌دهی Static بی‌فایده بوده و باید به سراغ راهکاری جدید برای انجام این کار برویم. راهکاری که در این فصل قصد بررسی آنرا داریم **Service Discovery** است.

به طور کلی به دو صورت می‌توانیم این روال را پیاده‌سازی کنیم که عبارتنداز **Server-Side Discovery** و **Client-Side Discovery**، که در ادامه با این دو روش آشنا خواهیم شد.

۱_ آشنایی با Service Registry

همانطور که قبلاً بیان کردیم تا پیش از این با توجه به ایستا بودن آدرس‌ها، نگهداری آن‌ها در فایل Config انجام می‌شد، اما با توجه به تغییرات صورت گرفته، نیاز داریم که وظیفه نگهداری آدرس‌های مختلف از سرویس‌های مختلف را به صورت داینامیک انجام شود. این وظیفه توسط Service Registry انجام می‌شود.

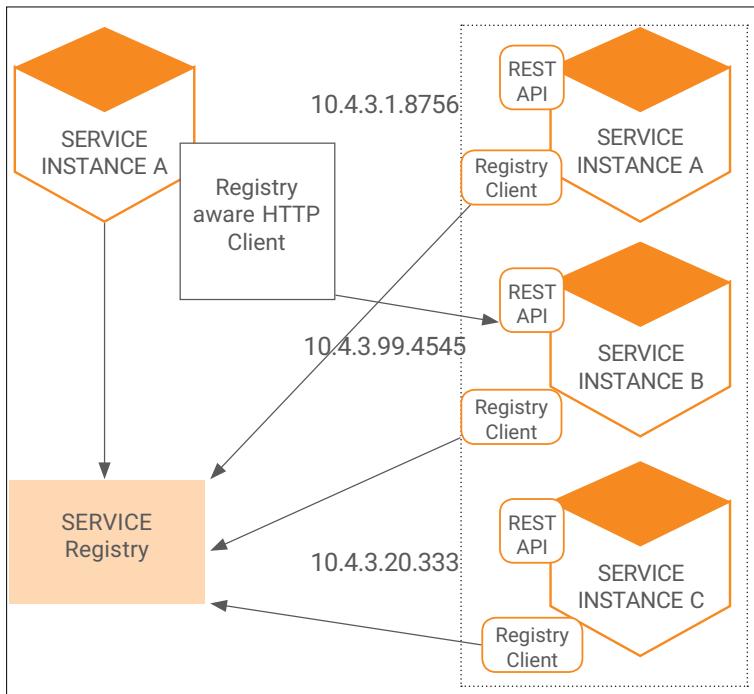
۲_ آشنایی با الگوی Client-Side Discovery

زمانی که از الگوی Client-Side Discovery استفاده می‌کنیم، وظیفه تشخیص محل سرویس‌ها و برقراری توازن در ارسال درخواست به نسخه‌های مختلف یک سرویس، به عهده کلاینت است. در این روش ابتدا کلاینت یک Query روی Service Registry اجرا کرده و لیستی از نمونه‌های سرویس دلخواه و اکشی می‌کند، سپس با یک الگوریتم توزیع بار، نمونه‌ای از سرویس را انتخاب کرده و درخواست را برای آن ارسال می‌کند.

هر محصولی که برای عمل کردن نیاز به دفترچه راهنمای داشته باشد، محصولی شکست خورده است.

ایلان ماسک

در تصویر زیر نحوه انجام این کار را مشاهده می‌کنید.



آدرس نمونه‌های متفاوت از سرویس‌ها هنگام شروع به کار سرویس ثبت شده و با پایان یافتن اجرای یک سرویس آدرس آن نیز از Service Registry حذف می‌شود. در بازه‌های زمانی مشخصی برای بررسی وجود یا عدم وجود سرویس‌های ثبت شده بررسی می‌شود. این بازه‌های زمانی را اصطلاحاً Heartbeat می‌نامیم.

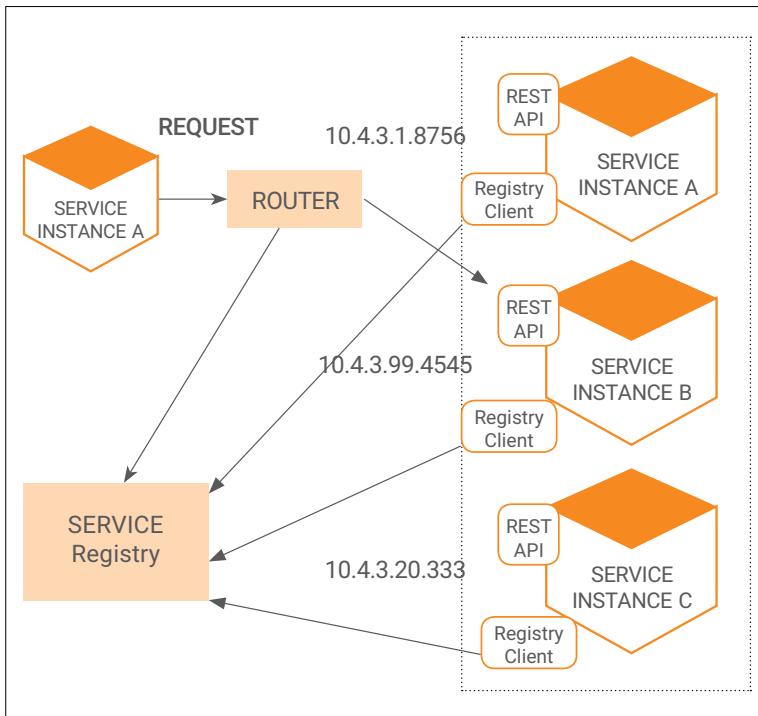
یک پیاده‌سازی مناسب از این الگو را در OSS Netflix می‌توانید مشاهده کنید. در این نمونه Netflix Eureka به عنوان یک Service Registry ایفای نقش می‌کند. Eureka برای ثبت، حذف و جستجوی آدرس سرویس‌ها تعدادی REST API در اختیار استفاده کنندگان قرار می‌دهد. برای پیاده‌سازی یک روال توزیع بار مناسب نیز می‌توانیم از Netflix Ribbon در کنار سایر ابزارها استفاده کنیم.

استفاده از این روش مزایا و معایب خاص خود را دارد که در ادامه بررسی می‌کنیم. بزرگترین مزیت این روش سادگی آن است. تنها کافیست آدرس Service Registry در اختیار کلاینت قرار بگیرد و کار تمام است. در این روش هر کلاینت می‌تواند الگوی توزیع بار اختصاصی خود را با توجه به شرایط اختصاصی خود پیاده‌سازی کند. بزرگترین عیب این روش هم می‌تواند وابستگی کامل کلاینت به Service Registry باشد. در نهایت اجبار به پیاده‌سازی این الگو به ازای هر کلاینت هم در صورتی که کلاینت‌های زیادی داشته باشیم می‌تواند خیلی سریع تبدیل به کابوس هر شب توسعه‌دهنگان شود.

حال که با پیاده‌سازی الگوی Service Discovery سمت کلاینت آشنا شدیم به سراغ پیاده‌سازی این الگو سمت سرور می‌رویم.

۳_ آشنایی با الگوی Server-Side Discovery

روش دیگر پیاده‌سازی الگوی Service Discovery، پیاده‌سازی آن سمت سرور بوده که می‌توانید در تصویر صفحه بعد مشاهده کنید:



در این روش، کلاینت درخواست‌های خود را برای یک مسیریاب ارسال می‌کند سپس مسیریاب Service Registry را برای یافتن آدرس نمونه‌های ثبت شده از سرویس هدف جستجو می‌کند. در پایان بعد از یافتن نمونه‌های سرویس با یک الگوریتم توزیع مناسب، نمونه‌ای از سرویس انتخاب شده و درخواست برای پردازش به آن سرویس ارسال می‌شود. ثبت و حذف سرویس‌ها در این روش هم، مانند پیاده‌سازی توسط کلاینت انجام می‌شود.

به عنوان نمونه‌ای از پیاده‌سازی یک مسیریاب می‌توان به AWS Elastic Load Balancing اشاره کرد. برای استفاده از این سیستم، درخواست‌های

از سرویس‌ها متفاوت درخواست‌ها بین آن‌ها توزیع می‌شود. برخی از محیط‌های توزیع مانند Kubernetes یک پروکسی ایجاد می‌کنند که مسئولیت Server-Side Load Balancer را انجام می‌دهند. در این روش کلاینت‌ها درخواست‌های خود را برای Proxy ارسال می‌کنند و Proxy به صورت هوشمند درخواست را برای سرویس‌های در دسترس ارسال می‌کند. پیاده‌سازی الگوی Service Discovery سمت سرور هم مانند هر روش دیگری مزایا و معایب خودش را دارد که در ادامه بررسی خواهیم کرد.

۴- بررسی دقیق :Service Registry

همانطور که قبلاً بیان شد قلب الگوی Service Discovery، دیتابیسی آدرس‌ها یا Service Registry است. Service Registry دیتابیسی است که آدرس دقیق هر نمونه از سرویس‌ها را نگهداری می‌کند. با توجه به نقش مهم این بخش، Service Registry باید همیشه در دسترس باشد و همیشه به روز باشد. برای افزایش بهره‌وری، کلاینت‌ها می‌توانند آدرس‌های به دست‌آمده از Service Registry را کش کنند، اما با توجه به اینکه این اطلاعات دائماً به روز می‌شود، باید روالی برای به روز رسانی داده‌های کش شده جهت جلوگیری از بروز مشکل سمت کلاینت در نظر گرفته شود.

همانطور که قبلاً ذکر شد Eureka یک نمونه خوب از پیاده‌سازی است. این ابزار یک REST API قوی برای ثبت و مدیریت آدرس سرویس‌ها ارائه می‌کند. برای ثبت یک آدرس جدید باید

یک درخواست POST به Eureka ارسال کنیم. برای پیاده‌سازی الگوی Heartbeat و زنده ماندن سرویس به صورت پیش‌فرض سرویس‌ها باید هر ۳۰ ثانیه یک درخواست PUT برای Eureka ارسال کنند. در صورتی که مدتی بیش از ۳۰ ثانیه بگذرد و درخواست PUT ارسال نشود یا اینکه یک درخواست DELETE برای Eureka ارسال شود، سرویس از پایگاه آن حذف می‌شود. اگر با سرویس‌های REST آشنا باشید احتمالاً تا الان حدس زده‌اید که برای دریافت یک آدرس باید یک درخواست GET برای Eureka ارسال کنیم. با توجه به اهمیت Service Registry و نیاز به دردسترس بودن همیشگی این سرویس، می‌توان از روال‌های Clustering همراه با Eureka استفاده کرد. پیشنهاد می‌کنم حتماً برای آشنایی بیشتر با Eureka به مستندات آن مراجعه کنید اما اگر به هر دلیلی تمایل داشتید از ابزار دیگری استفاده کنید می‌توانید از موارد ذیل استفاده کنید:

اولین گزینه Etcд است. یک پایگاه داده Key-Value و توزیع شده که برای نگهداری تنظیمات و Service Discovery استفاده می‌شود. به عنوان یکی از پروژه‌های مهمی که از این ابزار استفاده می‌کند Kubernetes را می‌توان نام برد.

به عنوان دومین ابزار مهم در این دسته می‌توان از Consul نام برد. این سیستم‌ها مانند بقیه موارد این دسته امکان ثبت و مدیریت سرویس‌ها را به کمک API فراهم می‌کند. به کمک این ابزار و امکانات آن می‌توانید وضعیت سلامتی و به روز بودن سرویس‌های خود را نیز تحت نظر قرار دهید.

و آخرین ابزاری که در این قسمت قابل بررسی است Apache ZooKeeper نام دارد. این ابزار هم یکی از موفق‌ترین نمونه‌های این دسته

است که ابتدا به عنوان بخشی از پروژه Hadoop معرفی شد و با گذشت زمان به سرویسی مجزا تبدیل شد.

همانطور که قبل ام عنوان شد اگر از برخی زیرساختها مثل Kuberne-tes یا AWS استفاده کنیم، نیازی به ابزار جداگانه برای Service Registry نداریم این الگو به صورت توکار توسط زیرساخت پشتیبانی می‌شود.

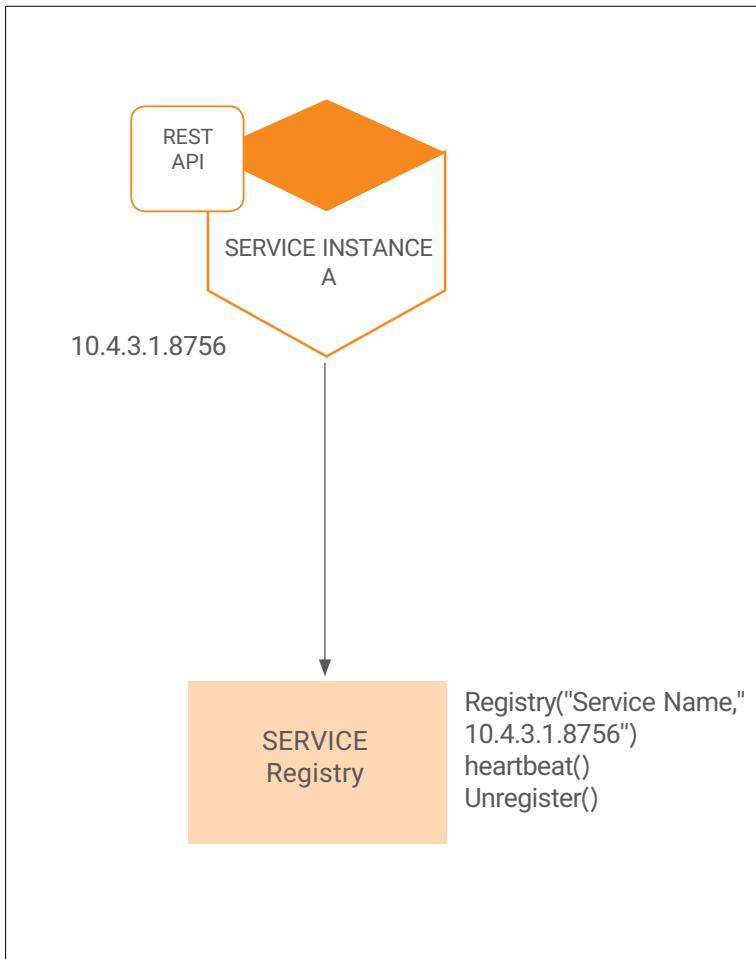
حال که با کلیات Service Registry و ابزارهای آن آشنا شدیم باید نگاهی به انواع روش‌های ثبت و مدیریت آدرس سرویس‌ها بیاندازیم.

۵_ انواع روش‌های مدیریت سرویس‌ها:

همانطور که قبل توضیح داده شد، باید امکانی داشته باشیم که نمونه‌های جدید سرویس‌ها را در Service Registry ثبت کنیم و در موقعی که نیاز داریم نمونه‌های ثبت شده را از دیتابیس حذف کنیم. این کارها را به روش‌های متفاوتی می‌توانیم انجام دهیم. اولین گزینه ثبت و مدیریت آدرس نمونه سرویس‌ها توسط خود سرویس‌ها است که به این روش Self Registration گفته می‌شود. اما این کار می‌تواند توسط یک سرویس Third-Party Registration دیگر انجام شود که به این روش نیز اصطلاحا گفته می‌شود. باید باهم این روش‌ها را دقیق‌تر بررسی کنیم.

۱_۵_ الگوی Self-Registration

هنگامی که از این الگو استفاده می‌کنیم، هر سرویس مسئول ثبت و حذف آدرس خود در Service Registry است. در صورت نیاز درخواست‌های Heartbeat نیز باید توسط خود سرویس‌ها ارسال شود تا از حذف ناخواسته سرویس از Service Registry جلوگیری کند.

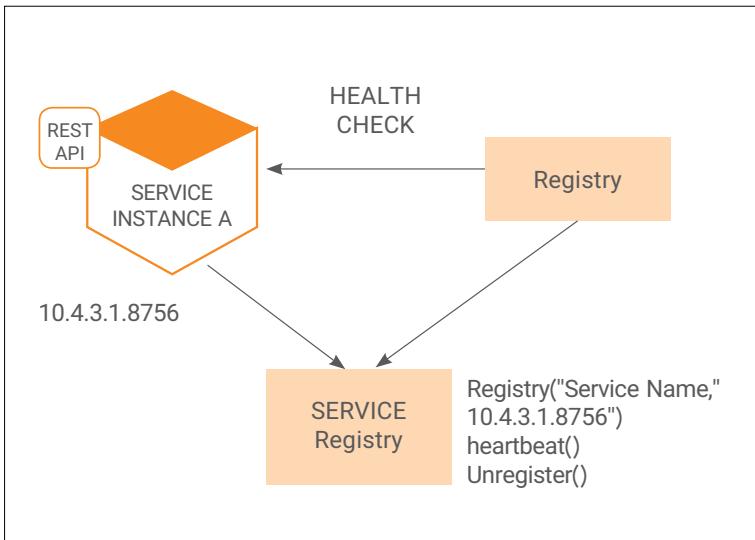


نمونه خوبی از پیاده‌سازی این الگو را در Netflix OSS Eureka Client می‌توانید مشاهده کنید. Eureka Client تمام کارهای مربوط به ثبت، حذف و مدیریت نمونه‌های سرویس‌ها را برای ما انجام می‌دهد.

یکی از مزایای استفاده از این روش سادگی پیاده‌سازی آن است. در این روش دیگر نیازی به استفاده از ابزار دیگری نیست و هر سرویس خود مسئول مدیریت نمونه‌های خودش می‌باشد. هر چند در کنار این سادگی باید بپذیریم که سرویس‌های ما به جز وظیفه‌ای که در نرم‌افزار ما به عهده دارند باید وظیفه دیگری را نیز به عهده بگیرند. پیاده‌سازی تکراری روال مدیریت نمونه سرویس‌ها نیز یکی دیگر از مشکلات پیاده‌سازی به این روش است. ضعف دیگر این روش پیاده‌سازی هم این است که به ازای هر زبان برنامه‌نویسی و فریمورکی که استفاده می‌کنیم باید بتوانیم این الگو را پیاده‌سازی کنیم.

۵_۲_الگوی Third-Party Registration

هنگامی که از این روش استفاده می‌کنیم دیگر نیازی نیست که هر سرویس به صورت خودمختار و مجزا اقدام به مدیریت نمونه‌های خود بکند. بلکه در این روش یک سرویس دیگر به سیستم اضافه می‌شود که به آن Service Registrar می‌گوییم. در این روش یا سرویس‌ها باید در زمان‌هایی خاص Event‌هایی را ایجاد کنند که Service Registrar مسئول مدیریت این Event‌ها است و یا اینکه زیرساخت باید به Service Registrar معرفی شود و به طور دائم زیرساخت توسط Service Registrar برای اجرای نمونه‌ای جدید از سرویس یا توقف یک نمونه از سرویس بررسی شود. در نهایت به یکی از روش‌های بالا Registrar از وضعیت سرویس‌ها مطلع می‌شود و نسبت و ثبت و یا حذف سرویس‌ها اقدام می‌کند.



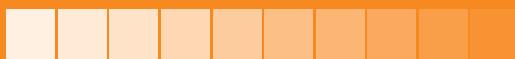
پروژه Registrator یکی از نمونه‌های پیاده‌سازی این روش است. این ابزار به صورت خودکار سرویس‌هایی که توسط Docker مدیریت می‌شوند را ثبت و حذف می‌کند. این ابزار Service Registry‌ها را پشتیبانی می‌کند. متعددی از جمله Etcd و Consul را پشتیبانی می‌کند. اما این الگو هم مانند هر الگوی دیگری مزايا و معابي دارد که باید با آن‌ها آشنا باشيم که در موقع لزوم بتوانيم به درستی ابزار مورد نياز خود را انتخاب کنيم. جداسازی سرویس‌ها از هر دلمشغولی به جز عملکرد مورد نياز خودشان بزرگترین مزیت پیاده‌سازی اين الگو است. دیگر نيازی نiest که به ازاي هر سرویس منطق مربوط به ثبت و مدیریت سرویس‌ها را پیاده‌سازی کنيم. به جاي همه اين کارهای اضافه، انجام عملیات ثبت و مدیریت نمونه‌های سرویس‌ها به صورت

مرکزی و خودکار انجام می‌شود. با تمام این مزایایی که ذکر شد باید دقت داشته باشیم زمانی که یک سرویس و ابزار جدید به سیستم اضافه می‌شود که مثل سایر سرویس‌ها نیاز به نگهداری و مدیریت دارد و زیاد شدن این سرویس‌های خدماتی می‌تواند خیلی زود تبدیل به هیولایی غیرقابل کنترل شود.

۶. خلاصه:

در میکروسرویس‌ها نمونه‌های نامشخصی از هر سرویس امکان اجرا دارد. با توجه به زیرساخت‌هایی که در میکروسرویس‌ها استفاده می‌شود، آدرس‌های آن‌ها نیز دائمًا احتمال تغییر دارد و دیگر نگهداری ایستای آدرس‌ها کارساز نخواهد بود و باید این مشکل را به شکلی حل کنیم که در این فصل به بررسی روش حل همین مشکل پرداختیم. در قسمت بعد راجع به مدیریت داده‌ها در میکروسرویس‌ها صحبت خواهیم کرد.

فصل ششم: مدیریت داده‌ها در میکروسرویس‌ها



- مشکلات داده‌های توزیع شده در میکروسرویس‌ها
- توسعه بر مبانی Event ها
- ویژگی Automicity هنگام استفاده از Event ها
- ایجاد و ارسال Event ها در تراکنش‌های محلی
- کاوش در Transaction Log
- استفاده از Event Sourcing

مقدمه: مشکلات داده‌های توزیع شده در میکروسرویس‌ها

به طور معمول هنگامی که نرم‌افزاری را به روش Monolithic توسعه می‌دهیم از دیتابیس‌های رابطه‌ای استفاده می‌کنیم که این دیتابیس‌ها به صورت توکار ویژگی ACID را برای تراکنش‌ها به همراه می‌آورند. ACID که مخفف چهار کلمه Conistency، Isolation، Durability و Automicity می‌باشد ضمانت‌هایی را به ما می‌دهد که در ادامه مختصراً در موردشان صحبت خواهیم کرد.

اول ■ خاصیت همه یا هیچ نیز معروف است تضمینی **Automicity**

می‌کند که در یک تراکنش اگر چندین عملیات انجام شود، یا همه آن‌ها کامل انجام می‌شوند یا هیچ کدام اجرا نخواهند شد. برای مثال فرض کنید در یک پایگاه داده لازم است اطلاعات یک جدول به روز رسانی شده و اطلاعات جدولی دیگر حذف شود و نتیجه انجام این دو عملیات در جدول سومی ثبت شود. در این شرایط یا هر سه عملیات کامل انجام می‌شود یا اگر به هر دلیلی هر کدام از اعمال انجام نشود، کل روال لغو می‌شود و اگر فرض کنیم در مرحله حذف به مشکل برخورد کرده باشیم، اطلاعات به روز شده نیز مجدد به حالت اولیه خود باز می‌گردند.

دوم ■ Consistency: به اصطلاح سازگاری به ما تضمین می‌دهد که

اگر یک تراکنش کامل انجام شود قطعاً پایگاه داده از یک وضعیت صحیح به وضعیت صحیح دیگری خواهد رفت. برای مثلاً فرض کنید

که در حساب شماره ۱ مبلغ ۱۰۰ تومان پول وجود دارد و در حساب شماره ۲ هم ۲۰۰ تومان پول وجود دارد و در مجموع ۳۰۰ تومان پول در بانک داریم. حال اگر مبلغ ۵۰ تومان از حساب یک به حساب دو منتقل شود بعد از تکمیل تراکنش قطعاً در حساب ۱ مبلغ ۵۰ تومان و در حساب شماره ۲ مبلغ ۲۵۰ تومان وجود خواهد داشت و مبلغ کل کماکان ۳۰۰ تومان باقی می‌ماند و ممکن نیست مبلغ کل برای مثلاً بعد از جابجایی مثلًا ۳۲۰ تومان بشود.

■ سوم Isolation: تصمیم می‌کند که در صورتی که چندین تراکنش در پایگاه داده همزمان اجرا شوند، این تراکنش‌ها از حضور یکدیگر اطلاع نداشته باشند و اجرای همزمان چند تراکنش تاثیر منفی در روند اجرای سایرین نداشته باشد. اگر هم نیاز باشد تراکنش‌ها تاثیری بر یکدیگر داشته باشند این تاثیر به شکلی است که در پایان با بررسی نتیجه گویا دستورات به صورت سریال انجام شده‌اند نه همزمان.

■ چهارم Durability: به قانون پایداری معروف است به ما اطمینان می‌دهد بعد از پایان یک تراکنش نتیجه اجرای آن تحت هیچ شرایطی به طور ناخواسته از پایگاه داده حذف نخواهد شد و قاعده‌تا Undo نمی‌شود. در نتیجه این ویژگی جذاب در یک برنامه Monolithic به سادگی می‌توانیم یک تراکنش را شروع کنیم و تعداد زیادی عملیات انجام دهیم و در پایان تراکنش را Commit کنیم و خیالمان از همه اتفاقات پایگاه داده راحت باشد.

ویژگی جذاب دیگری که در دیتابیس‌های رابطه‌ای در اختیار داریم

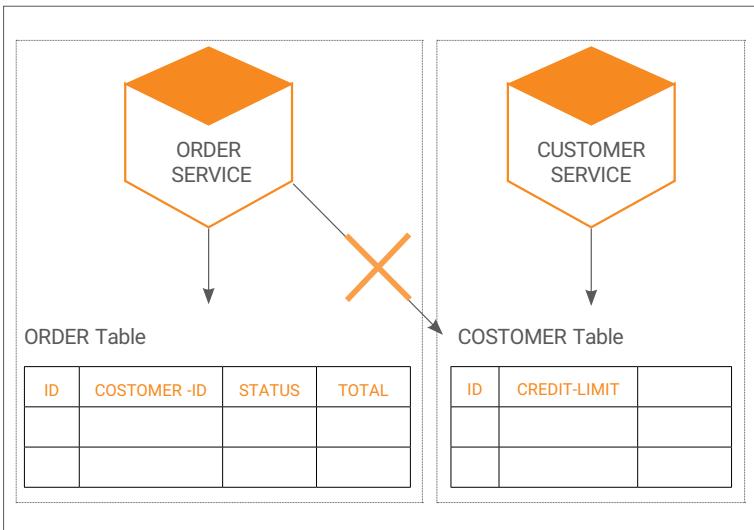
زبان SQL است. شاید استفاده زیاد از این زبان ارزش‌ها و مزایای فوق العاده آن را کمرنگ کرده باشد، اما با کمی دقت خواهیم دید به کمک این زبان، گویی چوب جادو در دست داریم. با چند دستور داده‌ها را از چند جدول مختلف می‌خوانیم و ترکیب می‌کنیم و خلاصه‌سازی می‌کنیم و نتیجه را مشاهده می‌کنیم و در پشت صحنه Query Planner وظیفه تفسیر، بهینه‌سازی و برنامه‌ریزی برای اجرای دستورات ما را به عهده می‌گیرد. نیازی نیست که نگران هیچ یک از این مراحل باشیم، فقط کافیست چوب جادو را تکان دهیم و ورد مخصوص را بخوانیم.

متاسفانه اگر از روش میکروسرویس برای توسعه نرم‌افزارهای خود استفاده کنیم بسیاری از این ویژگی‌ها را از دست خواهیم داد. هنگام توسعه میکروسرویس‌ها هر سرویس باید کل داده‌های خود را به صورت مستقل و در پایگاه داده‌ای مستقل در اختیار داشته باشد. اگر سرویسی به داده‌های سرویس دیگر نیاز داشته باشد، فقط از طریق API‌های ارائه شده اجازه دسترسی به داده‌های آن سرویس را خواهد داشت. این روش توسعه به ما اطمینان می‌دهد که هر کدام از سرویس‌ها به صورت مستقل امکان ارائه خدمات را خواهند داشت. در کنار این استقلال، با توجه به اینکه هر سرویس می‌تواند با ابزارها و فریمورک‌های خاص خود توسعه داده شود این امکان وجود دارد که برای هر سرویس و با توجه به نیازهای خاص آن از DB Engine مختلفی استفاده شود. ممکن است در یک سرویس از یک NOSQL خاص استفاده شود و سرویسی دیگر از یک پایگاه داده رابطه‌ای برای نگهداری داده‌های خود استفاده کند. برای مثال سرویسی که نیاز به جستجوی متن دارد ممکن است از Elastic Search استفاده کند.

در سرویسی دیگر ممکن است نیاز به نگهداری روابط بین موجودیت‌ها داشته باشیم پس استفاده از یک گراف دیتابیس مثل Neo4J کار توسعه را ساده‌تر خواهد کرد. این ترکیب DB Engine‌های مختلف در سیستم را اصطلاحاً Polyglot Persistence می‌نامیم.

هنگامی که از این روش‌های نگهداری داده‌ها استفاده می‌کنیم مزایای زیادی مثل توزیع پذیری، عدم وابستگی، بهره‌وری بالاتر و ... را به دست خواهیم آورد. در کنار این مزایا اما چالش‌های زیادی برای نگهداری این داده‌های توزیع شده پیش رو خواهیم داشت.

اولين و شايد بزرگترین چالش در اين راه مديريت تراكنش‌هایی است که انجام مراحل و نگهداری داده‌های آن در چند سرویس مختلف انجام می‌شود. برای مثال فرض کنید يك سیستم ثبت سفاروش آنلاین داریم. در سرویس مدیریت مشتریان مشخصات مشتری و اعتبار آن‌ها نگهداری می‌شود. در سرویس سفارشات نیاز است داده‌های مشتری و اعتبار آن‌ها پیش از ثبت سفارش اعتبار سنجی شود. در يك برنامه Monolith-ic به سادگی از جدولی از پایگاه داده کوئری می‌گیریم اما در يك برنامه میکروسرویس انجام این کار ممکن نیست. چگونگی جدا بودن اطلاعات این دو سرویس را در تصویر صفحه بعد مشاهده می‌کنید.



همانطور که در تصویر مشاهده می‌کنید، سیستم سفارشات اجازه دسترسی مستقیم به داده‌های سیستم مشتری‌ها را ندارد و تنها با استفاده از API‌های این سرویس می‌تواند از داده‌های آن استفاده کند. در مثال بالا سرویس سفارشات تنها نیاز به خواندن اطلاعات سرویس مشتری‌ها دارد. مشکل زمانی مشخص‌تر می‌شود که نیاز به تغییر داده‌های هر دو سرویس به صورت همزمان داشته باشیم. در همین مثال فرض کنید بعد از ثبت یک سفارش مناسب با رقم سفارش نیاز باشد که اطلاعات اعتبار مشتری نیز به روز شود. در چنین شرایطی به ناچار باید از تراکنش‌های توزیع شده یا اصطلاحاً Two-Phase Commit استفاده کنیم اما در صورتی که با تئوری CAP آشنا باشید می‌دانید که باید بین Consistency و Availability یکی را انتخاب کنید که معمولاً برنده این رقابت Availability است. با توجه به اینکه

برخی از NOSQL‌ها از Two-Phase Commit پشتیبانی نمی‌کنند و معمولاً مهم‌تر از Consistency Availability برای این مهم پیدا کنیم.

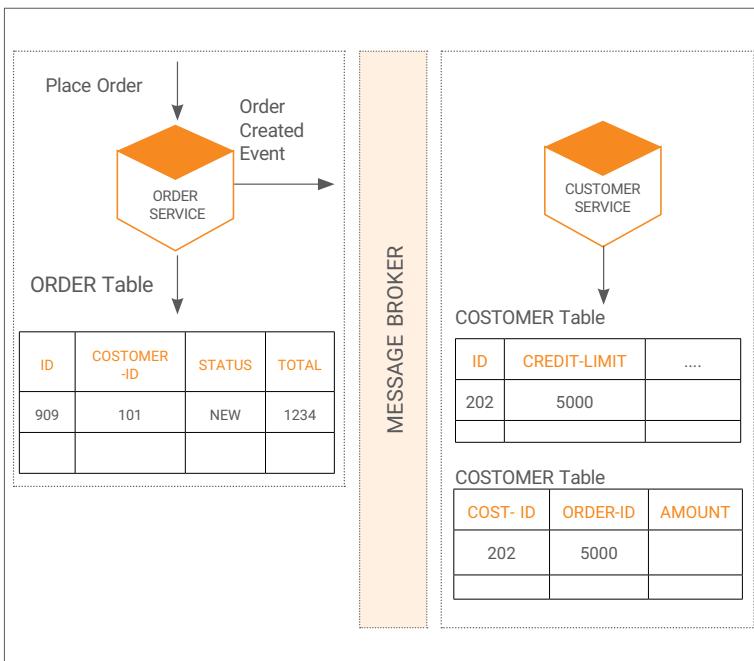
چالش دوم هنگامی معلوم می‌شود که نیاز داشته اطلاعاتی را از دو سرویس مختلف به دست بیاوریم. با کمی بررسی در مثال بالا کاملاً واضح است که نیاز داریم در بخشی از برنامه، اطلاعات مشتریان و سفارشات آن‌ها همزمان نمایش داده شود. در صورتی که فقط بر اساس API‌های ارائه شده بخواهیم داده‌ها را نمایش دهیم باید در Application یک Join ایجاد کنیم و داده‌های هر مشتری را به سفارشات مشتری متصل کنیم.

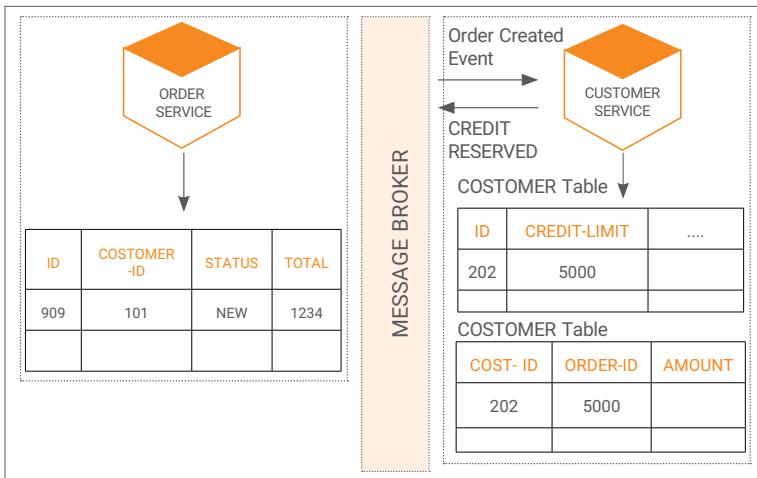
توسعه بر مبانی Event‌ها:

در بسیاری از برنامه‌ها راهکار مشکل استفاده از Event‌ها است. در این روش هر زمانی که یکی از داده‌های مهم سرویس تغییر می‌کند یک Event در سیستم ایجاد و ارسال می‌شود و سایر سرویس‌های موجود در سیستم می‌توانند در صورت نیاز این Event را مدیریت کنند و داده‌های داخلی خود را با توجه به این Event به روز کنند. این تغییر داده‌ها خود می‌تواند موجب ایجاد چندین Event دیگر در سیستم شود.

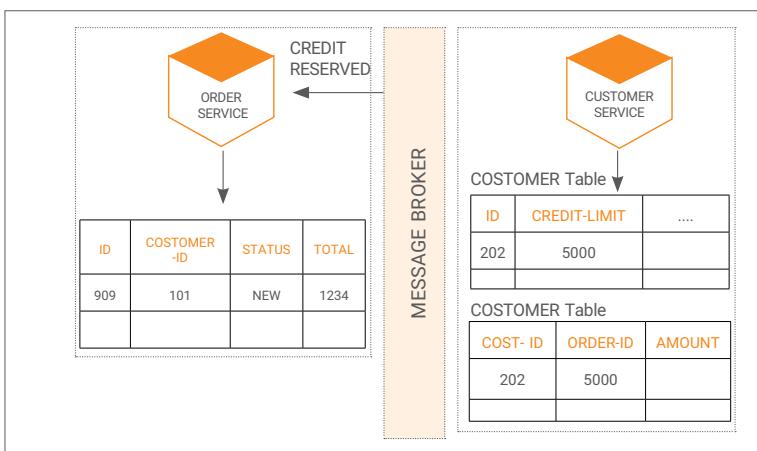
از Event‌ها برای پیاده‌سازی تراکنش‌های اپلیکیشن نیز می‌توان بهره برد. هر تراکنش شامل چند مرحله است. در هر مرحله در یک میکروسرویس داده‌هایی به روز می‌شوند و با تکمیل این مرحله یک Event ایجاد می‌شود که موجب فعال شدن مرحله بعد می‌شود و در نهایت با اتمام همه مراحل تراکنش به پایان می‌رسد.

در تصویر زیر روال بررسی اعتبار لازم برای مشتری را در سیستم ثبت سفارش مشاهده می کنید. در این سیستم میکروسرویس ها از یک Message Broker برای انتقال Event ها استفاده می کنند.





در پاسخ به رخداد قبلی، سرویس مدیریت کاربران مقدار مورد نیاز از اعتبار کاربر را رزرو کرده و Event متناسب با این تغییر را ارسال می‌کند.

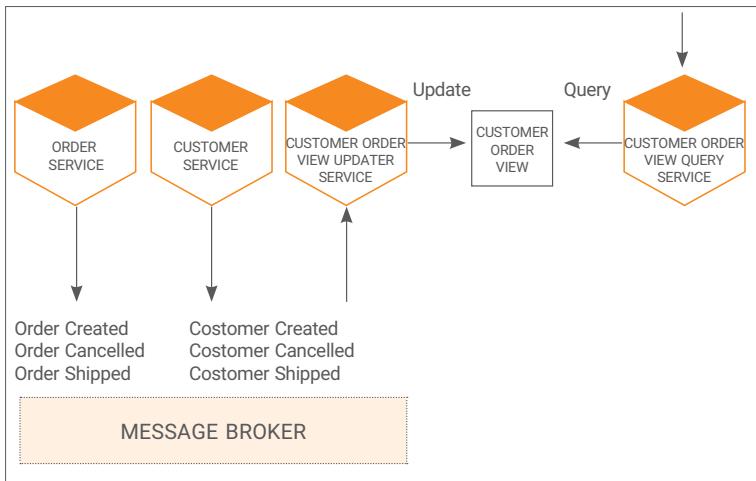


سرویس ثبت سفارش Event ارسالی از مدیریت مشتری را دریافت کرده و با توجه به داده‌های آن وضعیت سفارش را به OPEN تغییر می‌دهد.

البته در سناریوهای پیچیده‌تر و واقعی ممکن است مراحل و تغییر وضعیت‌های بیش از آنچه در این مثال دیدیم داشته باشد. مثلاً اینکه با رزرو اعتبار کاربر باید موجودی انبار بابت سفارش هم رزرو شود و ... که برای جلوگیری از پیچیده‌تر شدن مثال به همین اندازه اکتفا می‌کنیم.

در این روش در ابتدا هر سرویس به صورت Atomic داده‌های داخلی خود را تغییر می‌دهد و سپس یک Event در سیستم ایجاد می‌کند. در ادامه Message Broker تضمین می‌کند که این Event به تمامی سرویس‌ها برسد و در سومین مرحله هر سرویسی که این Event را دریافت کرده می‌تواند عملیات اختصاصی خود را به صورت Atomic انجام دهد و این چرخه را ادامه دهد. دقت کنید که به این روش نمی‌توانیم به خاصیت ACID برسیم و تنها چیزی که در این سیستم می‌شود صحت داده‌ها بعد از بازه‌ای از زمان است که اصطلاحاً به این حالت Eventual Consistency می‌گویند.

شما از این Event‌ها می‌توانید برای مدیریت داده‌های Materialized View نیز استفاده کنید. View‌ها که شامل داده‌های از پیش Join شده و آماده مصرف هستند، در پایگاه داده‌هایی نگهداری می‌شوند که توسط همه یا برخی سرویس‌های حاضر در سیستم به صورت مشترک قابل دسترس است. سرویسی که مسئول به روز رسانی داده‌های View است تقریباً تمامی Event‌های موجود در سیستم را دریافت و مدیریت می‌کند و با توجه به هر Event داده‌های صفر تا چندین View را به روز می‌کند. در شکل بعد سرویسی را مشاهده می‌کنید که مسئول به روز رسانی View مربوط به سفارشات مشتری‌ها است.



هنگامی که هر کدام از Event‌های مرتبط با Order یا Customer در سیستم رخ می‌دهد، این سرویس داده‌های این View را به روز می‌کند. داده‌های این View را می‌توانید در یک دیتابیس مبتنی بر سند مثل MongoDB نگهداری کنید و برای هر کاربر یک سند ثبت کنید که داخل خود آرایه ای از سفارشات دارد.

توسعه سیستم‌ها به این روش مزایا و معایبی در پی خواهد داشت. توانایی ایجاد و مدیریت تراکنش‌هایی در سطح چندین سرویس و در نهایت دستیابی به Eventually Consistency یکی از بزرگترین مزایایی این روش توسعه است. توانایی ایجاد و مدیریت Materialized View‌ها و OLAP‌های تقریباً همیشه به روز از دیگر ویژگی‌های برتر این روش توسعه است.

در مقابل این مزایا پیچیدگی‌های فنی و غیرفنی پیاده‌سازی این روش یکی از بزرگترین معایب این روش توسعه است. نیاز به پیاده‌سازی

روال‌هایی برای Rollback کردن تراکنش‌هایی که به این روش پیاده‌سازی می‌شوند کاری سخت و زمانگیر است. ضمن اینکه در این سیستم سرویس‌ها باید توانایی کارکردن با داده‌هایی را داشته باشند که تراکنش آن‌ها هنوز به پایان نرسیده است. در زمانی که هنوز یک تراکنش به پایان نرسیده سرویس‌هایی که از Materialized View‌ها استفاده می‌کنند نیز ممکن است داده‌های غیر صحیح را مشاهده کنند. مشکل بعدی که ممکن است در سیستم بروز کند دریافت دوباره Event‌ها است. در واقع Message Broker تضمین می‌کند که Event‌ها حتماً یک بار به سرویس‌ها می‌رسد اما تضمینی بابت جلوگیری از دوباره ارسال کردن یک Event تکراری ارائه نمی‌کنند.

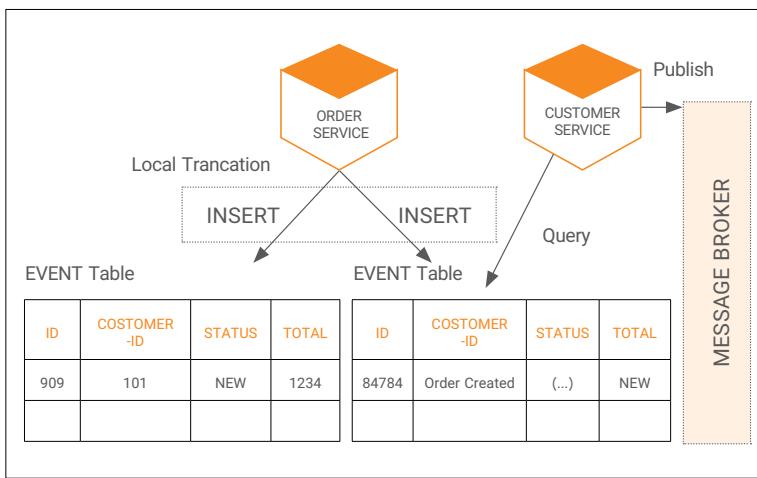
ویژگی Automicity Event‌ها

بیایید با هم کمی با جزئیات بیشتری به روal بالا دقیق کنیم. ابتدا یک سفارش ثبت می‌شود و ثبت یک Event در سیستم ایجاد و ارسال می‌گردد. تا این‌جای کار همه چیز خوب و عادی است اما اگر سفارش ثبت شود و سپس پیش از ایجاد و ارسال Event در سیستم دچار اختلال شویم چه اتفاقی می‌افتد؟ پاسخ ساده است داده‌های نامعتبر در سیستم ایجاد می‌شود.

■ ایجاد و ارسال Event‌ها در تراکنش‌های محلی:

یک راه برای به دست آوردن Automicity در این شرایط ایجاد یک پردازش چند مرحله‌ای و مدیریت مراحل به کمک یک تراکنش محلی است. تکنیکی که برای پیاده‌سازی این روش باید از آن استفاده کنیم

ایجاد یک جدول برای Event‌ها است. این جدول به عنوان یک Mesage Queue در سیستم عمل می‌کند که Event‌ها در آن ذخیره می‌شوند. هنگامی که یک موجودیت در سیستم تغییر می‌کند داده‌های Event معادل با این تغییر هم در این جدول ذخیره می‌شود. این ثبت اطلاعات همراه با تغییر داده‌های اصلی در یک تراکنش انجام می‌شود و به این روش، می‌توانیم مطمئن شویم که داده‌های Event‌ها از دست نمی‌رود و در صورت بروز مشکل داده‌ها در دیتابیس ثبت شده و با رفع مشکل سرویس، ارسال Event‌ها از سر گرفته می‌شود.

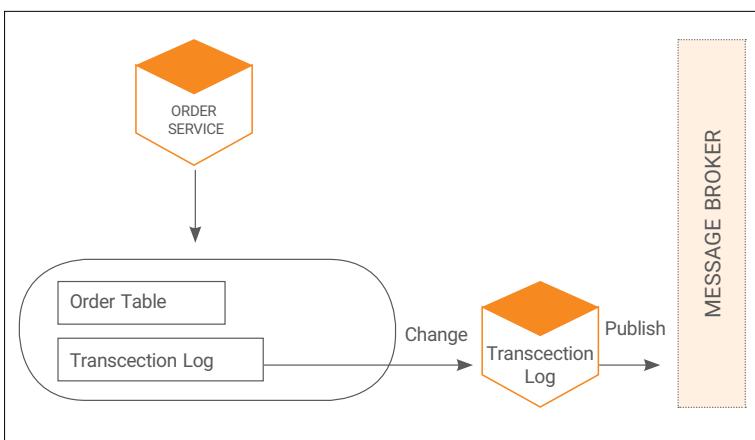


در این سیستم، ابتدا یک سفارش ثبت می‌شود و همزمان با آن یک Event هم با وضعیت ارسال نشده در جدول ثبت می‌شود. در ادامه داده‌ها از جدول Event دریافت شده و در سیستم ارسال می‌شود و در صورت صحت انجام کار، وضعیت Event به ارسال شده تغییر می‌کند و تراکنش با موفقیت به اتمام می‌رسد.

بزرگترین مزیت استفاده از این روش اطمینان از انتشار Event به ازای هر تراکنش داخلی بدون درگیر شدن با روالهای 2PC است. مشکلات پیاده‌سازی این روش موقع کار با بعضی NOSQL‌ها و احتمال فراموشی توسعه دهنده برای ذخیره Event یا مدیریت آن هم از مشکلات این روش است.

■ کاوش در Transaction Log : Transaction Log ■

راه بعدی برای دستیابی به Automicity بدون استفاده از 2PC تولید و ارسال Event‌ها به کمک Transaction Log می‌باشد. در این روش هنگامی که برنامه اصلی داده‌ای را تغییر می‌دهد به جز داده‌های اصلی، تغییری در Transaction Log هم ثبت می‌شود. حال در برنامه ای دیگر به سراغ این Transaction Log آمده و تغییرات را از روی این فایل به دست آورده و بر اساس آن‌ها در سیستم تولید و منتشر می‌کنیم. در تصویر بعد این حالت پیاده‌سازی را مشاهده می‌کنید.



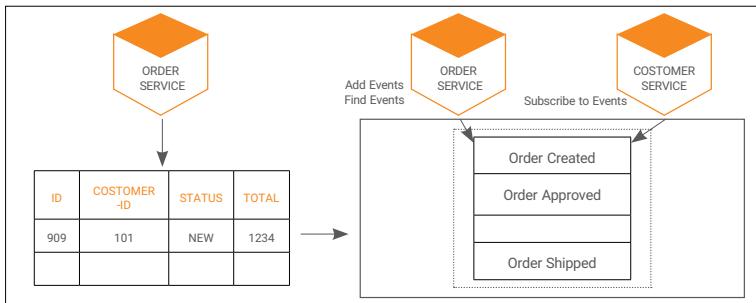
مثالی از این روش را می‌توانید در GitHub و در پروژه Databus مشاهده کنید. با استفاده از این سیستم شما می‌توانید لاغ‌های DB‌های مختلفی مثل اوراکل یا MySQL را پردازش کرده و با تشخیص تغییر داده‌ها Event‌هایی را در سیستم منتشر کنید. به عنوان مثال دیگری از این روش می‌توان به AWS Stream اشاره کرد. به کمک این مکانیزم تمامی تغییرات داده‌ها شامل ثبت، حذف و به روزرسانی به مدت ۲۴ ساعت در یک توالی زمانی نگهداری می‌شوند. حال برنامه مدیریت تراکنش‌ها می‌تواند داده‌های این جدول را خوانده و Event‌های مرتبط با آن‌ها را منتشر کند.

باز هم در پایان یک بخش باید به سراغ بررسی مزايا و معایب استفاده از یک روش توسعه برویم. تضمین ایجاد و انتشار Event بدون درگیر شدن با 2PC یکی از مزايا این روش است. به کمک این روش روای تولید و توزیع Event‌ها از چرخه برنامه اصلی خارج می‌شود و این کار می‌تواند به ساده شدن روای تولید و بهینه شدن انجام کارها کمک کند. بزرگترین ایراد این روش اما تفاوت در Transaction Log می‌باشد. بین دو نسخه متفاوت از یک DB Engine هم متفاوت است. حتی بعضاً ساختار و شرایط Event در موتورهای دیتابیس مختلف است. است. دشواری واکشی داده‌های سطح بالایی مثل اطلاعات یک Event از داده‌های سطح پایین ذخیره شده در لاغ‌ها نیز یکی دیگر از معایب این روش پیاده‌سازی است.

■ استفاده از Event Sourcing

در این روش برای دستیابی به Automicity از روشی کاملاً متفاوت با آنچه تاکنون دیده‌ایم استفاده می‌شود. در این روش به جای اینکه وضعیت یک Entity را ذخیره کنیم، وقایعی که روی این Entity رخ می‌دهد را ذخیره می‌کنیم. حال برای اینکه به وضعیت فعلی یکی Entity دسترسی داشته باشیم می‌توانیم وقایع رخ داده روی این Entity را از ابتدا اجرا کنیم تا مجدد Entity در وضعیت نهایی خود قرار بگیرید. در این روش هیچ داده‌ای تغییر نمی‌کند و اطلاعاتی نیز حذف نمی‌شود بلکه صرفاً وقایع روی Entity‌ها رخ می‌دهد و به ترتیب به پایگاه داده اضافه می‌شوند. از آنجایی که ذخیره شدن وقایع یک عملیات است، به صورت پیش فرض یک عملیات Atomic است.

برای درک بهتر این الگو باید نگاهی به مثال ثبت سفارش و پیاده‌سازی آن به کمک Event Sourcing بیاندازیم. در روش معمول به ازای هر سفارش یک ردیف داده در جدول Order ثبت می‌شود و اقلامی هم که در سفارش وجود دارند در جدول OrderLineItem یک ردیف معادل دارند. اما هنگام استفاده از Event Sourcing دیگر داده‌ها در جدول Order ثبت نمی‌شوند بلکه برای هر وضعیت سفارش مثل جدید، تایید شده، حمل شده یا کنسل شده یک ردیف در جدول Event‌ها ثبت می‌شود.



در این روش تمامی داده‌های رخدادها در یک پایگاه داده ثبت می‌شوند. داده‌های رخدادها را می‌توان در جداول و پایگاه‌های داده عادی نیز ذخیره کرد. اما دیتابیس‌های اختصاصی برای نگهداری اطلاعات Event‌ها مثل Event Store نیز وجود دارند که API‌های تخصصی برای ثبت و مدیریت داده‌های Event‌ها در اختیار قرار می‌دهند.

استفاده از این روش مزایای بسیاری داشته و مشکلات زیادی را حل می‌کند، برای مثال می‌توان به سادگی وضعیت Entity را در هر زمانی از گذشته تا به حال به دست آورد. می‌توان بدون درد سر روایی که اتفاق افتاده تا Entity در وضعیت فعلی قرار گرفته را مشاهده کرد. نیازی به نگهداری داده‌های اضافه برای دانستن دلیل و شرایط تغییر وضعیت ... Entity‌ها نیست و ...

اما در کنار این مزایا پیچیدگی پیاده‌سازی این روش، یا بهتر اگر بخواهیم بیان کنیم، نگرش جدیدی که باید به مقوله داده‌ها و نگهداری وضعیت Entity‌ها داشته باشیم یکی از عیوب‌های این روش است. ابراد دوم در این الگو این است که در صورتی که وقایع یک Entity زیاد باشد، زمان زیادی

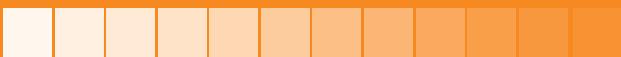
طول می‌کشد تا پردازش‌های لازم جهت رسیدن به وضعیت نهایی یک Entity انجام شود. سومین و آخرین ایراد این روش که در این مطلب بررسی می‌کنیم مشکل جستجو روی داده‌ها است. جستجوهای معمول را هم نمی‌توان با سادگی با این روش انجام داد و نیاز به پردازش زیادی داریم تا بتوانیم پاسخ یک Query ساده را از میان انبوهی از Event‌ها بیرون بکشیم.

با اینکه این الگو یک الگوی مستقل و قابل پیاده‌سازی به تنها‌ی است اما اغلب همراه با الگوی CQRS طراحی و پیاده‌سازی می‌گردد تا بخشی از مشکلات و ایرادات این الگو پوشش داده شود.

جمع‌بندی:

در معماری میکروسرویس‌ها هر میکروسرویس، دیتابیس و داده‌های اختصاصی خود را در اختیار دارد و می‌توانند به تنها‌ی و مستقل از هر سرویس دیگری خدمات خود را ارائه دهند. میکروسرویس‌های مختلف می‌توانند از DB‌های متفاوتی نیز استفاده کنند که قابلیت‌های ویژه‌ای برای آن سرویس خاص دارند که می‌تواند موجب بهینه یا ساده شدن انجام کار شود. اما این روش پیاده‌سازی چالش‌هایی را در مودر اشتراک و توزیع داده‌ها ایجاد می‌کند که توسعه دهنده‌گان باید راهکارهایی برای آن‌ها بیان‌دیشند. یکی از راهکارهای حل کردن این چالش‌ها استفاده از Event‌ها در سیستم است که در این مطلب این راهکار را با هم بررسی کردیم. در قسمت آینده در مورد استقرار میکروسرویس‌ها و چالش‌های آن صحبت خواهیم کرد.

فصل هفتم: امنیت در میکروسرویس



- تاریخچه، نحوه امن کردن نرم افزارهای یکپارچه
- مبانی کلیدی امنیت
- امنیت مرزهای نرم افزار
- امن کردن ارتباط سرویس به سرویس
- بحث و ارزیابی و نتیجه‌گیری

مقدمه:

استفاده از معماری میکروسرویس و ابزارها و زیرساخت‌های مرتبط با آن مزایای متعددی از قبیل، کاهش زمان تولید، کاهش هزینه شکست، بهبود کارایی، ارتقاء نرم‌افزارها به شکل مستمر، سازگاری با فناوری‌ها و زبان‌های برنامه‌نویسی متعدد و به روز، قابلیت تغییر سریع و استفاده از اجزا موجود را به ارمغان می‌آورد. همچنین باید به این موضوع دقت داشته باشیم که استفاده از این معماری و انتخاب آن به عنوان معماری اصلی نرم‌افزاری چالش‌های جدید بسیاری را به دنیای توسعه نرم‌افزار وارد می‌کند که یکی از آن‌ها تغییر در روش‌های برقراری امنیت و دسترسی به نرم‌افزار است.

در این فصل می‌خواهیم درباره مسائل مربوط به حوزه امنیت در توسعه نرم‌افزار صحبت کنیم و ببینیم که این چالش‌ها در زمان توسعه یکپارچه نرم‌افزار چگونه رفع می‌شده و نیازهای کاربران پاسخ داده می‌شده و در حال حاضر با تغییر نگرش تولید نرم‌افزار و تبدیل معماری نرم‌افزار به یک معماری توزیع چالش‌های موجود چگونه باید پیاده‌سازی شوند و این‌که آیا این چالش‌ها تغییری نسبت به حالت قبل داشته‌اند یا نه و تعداد آن‌ها ثابت است و صرفاً نحوه پیاده‌سازی هر مورد است که تغییر یافته‌است.

معماری میکروسرویس ساختاری را به نرم‌افزارها وارد می‌کند که به صورت ذاتی موجب ایجاد چالش‌های امنیتی می‌شود.

در نرم افزارهای یکپارچه، ارتباط بین اجزای درونی، داخل یک پردازش واحد اتفاق می‌افتد. بعنوان مثال درون یک ماشین مجازی جاوا^۱ تحت لوای معماری میکروسرویس، این اجزای درونی بطور مجزا طراحی شده‌اند (میکروسرویس‌های مستقل از هم)، و فراخوانی‌های درون پردازش بین اجزای درونی، تبدیل شده‌است به فراخوانی‌های از راه دور^۲. به عبارت دیگر، هر میکروسرویس بطور مستقل درخواست را مستقیماً می‌پذیرد و درگاه ورودی خودش را دارد.

به ازای چند درگاه ورودی در نرم افزارهای یکپارچه، اکنون شما تعداد خیلی زیادی درگاه ورودی دارید. حال هر چه تعداد این درگاه‌های ورودی بیشتر شود، گستردگی سطح حملات هم به طبع آن بیشتر می‌شود. این وضعیت، یکی از چالش‌های اساسی ایجاد طراحی امن برای میکروسرویس‌ها می‌باشد. هر درگاه ورودی برای هر میکروسرویس باید با استحکام برابر محافظت شود. امنیت کل سیستم، از استحکام ضعیفترین لینک آن بیشتر نیست.

در کنار این مشکل احراز هویت و تعیین سطوح دسترسی کاربران باید در سیستم‌های مختلفی انجام شود و راهکاری باید وجود داشته باشد که دائماً درگیر احراز هویت کاربران در بخش‌های مختلف نرم افزار نباشیم و بتوانیم کماکان احراز هویت و تعیین سطوح دسترسی را به صورت متمرکز در بخشی از نرم افزار انجام دهیم و بین قسمت‌های مختلف به اشتراک بگذاریم. در این فرایند به اشتراک‌گذاری باید به این نکته توجه کنیم که با

توجه به پرکاربرد بودن فرایند احراز هویت و تعیین سطوح دسترسی نباید کندی و اختلالی در این فرایند احساس شود.

در کنار مسائل روزمره‌ای که در حوزه امنیت در توسعه نرم‌افزار با آن سرو کار داریم، وقتی سراغ معماری میکروسرویس می‌آییم چالش‌های جدیدی نیز پیدا می‌کنیم. یکی از آن‌ها ابزارهایی است که به ناچار و برای پیاده‌سازی این معماری به سیستم وارد می‌شود و این ابزارها و تعیین سطح امنیتی آن‌ها یکی از مسائل مهمی است که با آن دست و پنجه نرم می‌کنیم. در کنار این مسائل درخواست‌هایی که در معماری یکپارچه بین دو یا چند ماژول نرم‌افزاری جابجا می‌شوند در حال حاضر باید در بستر شبکه و خارج از محیط داخلی برنامه ارسال شود و تامین امنیت این روال نیز می‌تواند ما را دچار مشکلات فراوانی نماید.

۲_ تاریخچه، نحوه امن کردن نرم‌افزارهای یکپارچه

یک نرم‌افزار یکپارچه تعداد کمی درگاه ورودی^۳ دارد. درگاه ورودی برای یک نرم‌افزار حکم درب یک ساختمان را دارد. همانطور که درب ورودی امکان وارد شدن به یک ساختمان را می‌دهد (احتمالاً بعد از بررسی‌های امنیتی)، درگاه ورودی یک اپلیکیشن هم امکان درخواست دادن^۴ را مهیا می‌کند.

یک نرم‌افزار تحت وب را تصور کنید (تصویر صفحه بعد را ببینید) که روی درگاه^۵ پیش‌فرض HTTP یعنی درگاه شماره ۸۰ یک سرور با آدرس

3-Entrypoint

4-Request

5-Port

در حال اجراست. درگاه ۸۰ سرور ۱۹۲.۱۶۸.۰.۱ روی این سرور به عنوان مدخل ورودی نرم‌افزار ما است. اگر همین نرم‌افزار درخواست‌های HTTPS روی درگاه ۴۴۳ همین سرور را هم بپذیرد، اکنون یک درگاه ورودی دیگر هم داریم. هرچقدر درگاه ورودی بیشتری داشته باشید، مکان‌های بیشتری برای نگرانی داریم. بعنوان مثال، زمانی که مرز بیشتری برای محافظت دارید، سربازهای بیشتری هم نیاز دارید، باید دیواری مستحکم برای تمام درگاه‌های ورودی بسازید. درگاه‌های ورودی بیشتر یک اپلیکیشن به معنای مرزهایی هستند که مورد حمله قرار می‌گیرند.

نرم‌افزارهای یکپارچه، معمولاً با استفاده از یک آدرس به دنیای بیرون از نرم‌افزار متصل می‌شوند و خدمات آن‌ها از طریق یک آدرس در دسترس قرار می‌گیرد.

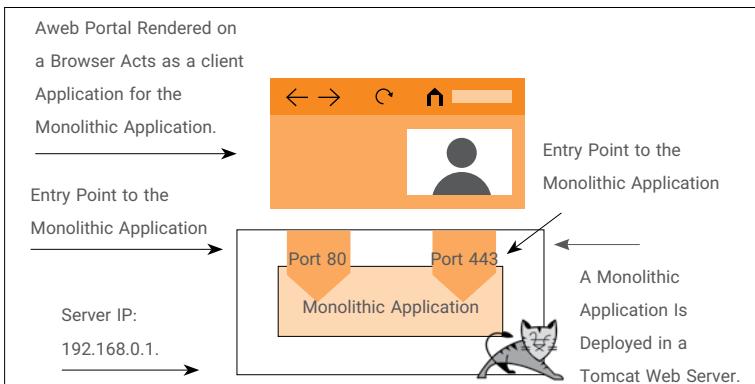
در بستر معمول جاوا (وب اپلیکیشن‌های نسخه‌ی سازمانی Java EE) که در تصویر زیر آمده است)، تمام درخواست‌های ورودی در سطح نرم‌افزار به واسطه‌ی Servlet Filter^۷ بررسی می‌شوند. در این غربالگری امنیتی بررسی می‌شود که آیا درخواست جاری مرتبط به یک نشست^۸ معتبر است و اگر نیست این درخواست را به سمت احراز هویت^۹ سوق دهد.

6-Component

7-چنانچه با servlet filter ها آشنایی ندارید، آنها را بازرس‌هایی تصور کنید که در وب اپلیکیشن تمام درخواست‌هایی که می‌آید را بازرسی می‌کنند.

8-Session

9-Authenticate



نمونه نرم افزار یکپارچه با دو درگاه ورودی

در مرحله بعد کنترل دسترسی‌های بیشتر بررسی می‌گردد که آیا طرف درخواست کننده^{۱۰} اجازه یا صلاحیت‌های^{۱۱} مورد نیاز برای آنچه که می‌خواهد را دارد یا نه. تمامی این کارها را بازرس بطور مرکزی انجام می‌دهد تا مطمئن شود درخواست‌های مشروع و مورد قبول به اجزای مربوطه برسد. اجزای درونی^{۱۲} اهمیتی به مشروعیت درخواست نمی‌دهند، و همواره چنین پیش‌فرضی دارند که اگر درخواستی که به این مرحله رسیده است حتماً موارد امنیتی آن بررسی شده‌است.

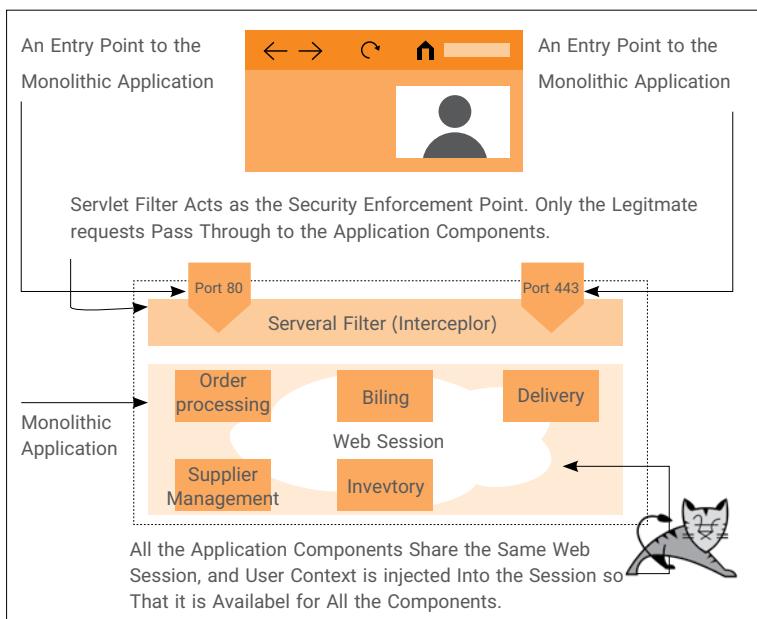
در مواردی که اجزا نیاز داشته باشند که بدانند طرف درخواست کننده (یا کاربر) کیست یا اطلاعاتی مرتبط با طرف درخواست کننده داشته باشند، چنین اطلاعاتی را می‌توانند از نشست وب که در سطح اپلیکیشن و برای تمام اجزا مشترک است، بازیابی کنند (تصویر صفحه بعد را مشاهده کنید). در فرایند غربالگری اولیه بعد از فرایند احراز هویت و احراز

10-Request party

11-permissions

12-Internal components

مجوزهای اطلاعات طرف درخواست کننده را به نشست و ب تزریق می‌کند. هنگامی که درخواست به لایه اپلیکیشن رسید، دیگر نباید نگرانی‌ای درخصوص ارتباط یک جزء با دیگری داشت. بعنوان مثال، زمانی‌که زیرسیستم Order Processing با زیرسیستم Inventory صحبت می‌کند، الزامی به بررسی‌های امنیتی بیشتر نیست (البته این به این معنا نیست که شما نمی‌توانید به ازای هر جزء، در سطح کامپوننت کنترل دسترسی داشته باشید). این عملیات فراخوانی‌های داخلی^{۱۳} هستند و در غالب اوقات شنود آن برای شخص ثالث کار دشواری خواهد بود.



اشتراك نشست توسط بازرس مرکزی برای اجزا داخلی سیستم

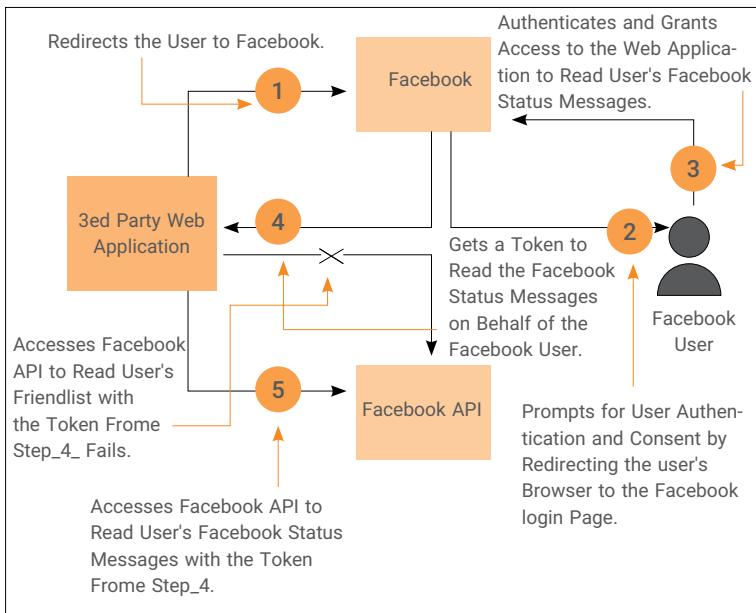
در اکثر نرم‌افزارهای یکپارچه، امنیت در یک بخش مرکزی قرار دارد و تک تک اجزا نیازی به بررسی‌های بیشتر ندارند مگر در موارد بخصوص بسته به نوع نیاز داخلی یک قسمت از برنامه. به همین دلیل مدل امنیتی نرم‌افزارهای یکپارچه به مراتب سر راست‌تر از میکروسرویس‌ها است.

۳- مبانی کلیدی امنیت

پاییندی به اصول در همه طرح‌های امنیتی مهم است. هیچ امنیت بی‌نقص و یا غیرقابل شکستنی وجود ندارد. باید این نکته را نیز در نظر بگیرید که میزان نگرانی شما در رابطه با امنیت صرفاً یک کار فنی نبوده و تاثیرات اقتصادی نیز دارد. برای مثال برای امن کردن یک گاراژ خالی استفاده از یک دزدگیر فوق پیشرفته بی‌فایده است. بسته به دارایی که از آن‌ها محافظت می‌کنیم، سطح امنیت نیز تعییر می‌کند. به طور یقین طرح امنیتی برای یک فروشگاه اینترنتی با یک نرم‌افزار مالی متفاوت است. پاییندی به اصول امنیتی مهم است. حتی اگر برخی تهدیدات امنیتی را پیش‌بینی نکنید، پیروی از اصول امنیتی به شما برای در امان ماندن از این تهدیدات کمک خواهد کرد. در این فصل، ما اصول امنیتی را گام به گام با شما بررسی می‌کنیم و به ارتباط آن‌ها با امنیت میکروسرویس‌ها خواهیم پرداخت.

۱_۳- جلوگیری از کلاهبرداری به کمک احراز هویت

شناسایی طرف درخواست کننده به منظور حفاظت از سیستم در برابر کلاهبرداری را احراز هویت می‌نامیم. طرف درخواست کننده می‌تواند یک سیستم باشد (یک میکروسرویس) یا یک سیستمی که توسط



اتصال سرویس‌ها به یکدیگر با و بدون عامل انسانی

کابر انسانی یا یک سیستم دیگر کنترل می‌شود (تصویر این صفحه را مشاهده کنید). این نوع دسترسی با دسترسی مستقیم کابر انسانی به یک میکروسرویس متفاوت است. پیش از ایجاد یک طرح امنیتی برای سیستمی که درباره آن صحبت کردیم، باید شرکت‌کنندگان را بشناسیم. روش احراز هویتی که استفاده می‌کنند وابسته به شرکت‌کنندگان است. اگر نگران دسترسی یک سیستم به میکروسرویس هستید که در پشت صحنه توسط یک کابر هدایت می‌شود باید به این موضوع فکر کنید که چگونه آن سیستم می‌تواند مانند یک کابر در سیستم احراز هویت شود. در عمل ممکن است یک وب اپلیکیشن داشته باشیم که کاربری انسانی در آن احراز هویت شده و آن سیستم به میکروسرویس متصل می‌شود.

در شرایطی، که یک سیستم درخواست دسترسی از طرف یک سیستم دیگر که قبل از آن یک عامل انسانی در آن احراز هویت شده است دارد، OAuth 2.0 استاندارد امنیتی مناسبی است.

برای احراز هویت کاربر انسانی به سیستم (برای مثال، یک وب اپلیکیشن)، در روش‌های مبتنی بر چند روش احراز هویت، باید درخواست نام کاربری و رمز عبور بهمراه عنصر دیگر برای احراز هویت داشته باشد (MFA). در اغلب موارد MFA یک تصمیم تجاری است که با توجه به میزان اهمیت دارایی‌های تجاری و یا میزان حساسیت داده‌هایی که می‌خواهید به کاربران منتقل کنید گرفته می‌شود. مشهورترین نوع استفاده از MFA کد عبور یکبار مصرف است (OTP) که معمولاً از طریق پیامک (SMS) ارسال می‌شود. شاید این راه در دنیای امنیت بهترین روش نباشد، اما پراستفاده‌ترین شکل از بکارگیری MFA می‌باشد، چرا که عمدتاً جمعیت کثیری از جهان به تلفن‌های موبایل دسترسی دارند که الزاماً نیازی هم نیست تلفن‌های هوشمند باشند. MFA به کاهش ۹۹/۹۹ درصدی رخنه‌های مربوط به حساب کاربری منجر شده است. انواع مطمئن‌تری از MFA وجود دارند که از علائم حیاتی، گواهینامه‌ها و Fast Identity Online (FIDO) استفاده می‌کند.

چندین راه برای احراز هویت یک سیستم وجود دارد که محبوب‌ترین گزینه‌ها استفاده از گواهینامه‌ها و JWT‌ها می‌باشد.

۳-۲_ جلوگیری از تغییر غیرمجاز داده‌ها به کمک یکپارچگی^{۱۰}

هنگامی که داده‌ها بین دو سرویس انتقال پیدا می‌کند، با توجه به

قدت کانال ارتباطی که مورد استفاده قرار می‌گیرد، ممکن است داده‌ها توسط فردی نفوذی در مسیر انتقال تغییر پیدا کند. برای مثال ممکن است پیامی که بین دو سیستم انتقال پیدا می‌کند مربوط به آدرس ارسال یک سفارش اینترنتی باشد و فرد نفوذی آدرس خود را با آدرس دریافت کننده حقیقی جایگزین کند. در حقیقت سیستم‌های مبتنی بر یکپارچگی راهکاری جهت جلوگیری از این دسترسی ارائه نمی‌کنند، بلکه راهکارهایی ایجاد می‌کنند که به کمک آن از تغییر داده‌ها در بین راه مطلع شوید.

یکی از راهکارهای معمول برای پیاده‌سازی یکپارچگی داده، امضا کردن آن‌ها است. هر داده‌ای که انتقال پیدا می‌کند به کمک ^{۱۵}TSL محافظت می‌شود. اگر از HTTPS برای انتقال داده‌ها بین سرویس‌ها استفاده کنید، پیام‌های شما به کمک TSL حفاظت می‌شود.

۳_۳_ انکار ناپذیری^{۱۶}

انکارناپذیری یکی از اصول امنیت نرم‌افزار است که مانع از عدم پذیرش مسئولیت کارهایی که انجام داده‌ایم می‌شود. در دنیای واقعی نیز چنین شرایطی وجود دارد. برای مثال فرض کنید زمانی که شما یک آپارتمان اجاره می‌کنید، برای مدت و هزینه‌ای معین، یک قرارداد امضا می‌کنید که با امضای آن قرارداد تصريح می‌کنید که به تمامی مفاد آن قرار داد پایبند هستید. شما نمی‌توانید اجاره ماهانه را پرداخت نکنید یا اگر بخواهید زودتر منزل را ترک کنید، یا باید اجاره ماههای باقی مانده را

15-Transport Layer Security

16-Nonrepudiation

پرداخت کنید یا ناچارید شخص جایگزینی برای اجاره منزل بیابید. این انکارناپذیری در دنیای واقعی است. در دنیای دیجیتال نیز امضا همین حکم را دارد و مانع از انکار حقایق می‌شود. فقط به جای امضای حقیقی با امضای دیجیتال سر و کار داریم.

فرض کنید یک فروشگاه اینترنتی دارید، اگر سفارشی در فروشگاه ثبت شود، میکروسرویس مدیریت سفارشات باید به سراغ میکروسرویس انبار برود و موجود آن کالا را کسر کرده و برای ارسال آماده کند. در صورتی که برای این تراکنش امضای دیجیتال در نظر گرفته شده باشد، در آینده میکروسرویس مدیریت سفارشات نمی‌تواند انجام این تراکنش را انکار کند. در روش امضای دیجیتال، مالک امضا یک کلید خصوصی دارد که تنها به کمک آن امکان ایجاد امضاهای کاملاً مشابه وجود دارد و سایرین به کمک کلید عمومی این امضا را تایید می‌کنند. حفظ و نگهداری کلید خصوصی بسیار اهمیت دارد.

۴_۳_ محramانگی^{۱۰} و جلوگیری از انتشار داده‌های خصوصی

هنگامی که درخواست ثبت یک سفارش از نرمافزار مشتری به میکروسرویس ثبت سفارشات ارسال می‌شود، شما توقع دارید که به جز بخش ثبت سفارشات سایر قسمت‌ها به اطلاعات در حال جابجاگی دسترسی نداشته باشند. با توجه به سطح امنیتی که کانال ارتباطی شما رعایت می‌کند، یک مت加وز می‌تواند به داده‌های در حال انتقال دسترسی داشته باشد. به جز داده‌هایی که روی کانال‌های ارتباطی

در حال انتقال هستند، داده‌های داخلی برنامه نیز باید از دسترسی غیرمجاز حفظ شوند. اگر یک فرد مت加وز به محل ذخیره‌سازی فایل‌ها یا پایگاه‌های داده شما دسترسی پیدا کند، به سادگی به تمامی داده‌های حیاتی کسب و کار شما دسترسی خواهد داشت، مگر اینکه آن‌ها را برای چنین روزهایی در سطح محترمانگی خوبی قرار داده باشید.

برای حفظ محترمانگی داده‌های در حال جابجایی معمولاً از رمزگذاری داده‌ها استفاده می‌کنند. یک روال رمزگذاری خوب این اطمینان را به ما می‌دهد که اطلاعاتی که رمزگاری شده‌است، تنها توسط اهدافی که قرار است به داده‌ها دسترسی داشته باشند قابل مشاهده است. اساساً TLS یکی از روش‌های اطمینان از محترمانگی هنگام انتقال داده‌هاست. اگر میکروسرویس‌های ما برای برقراری ارتباط از HTTPS استفاده کنند، می‌توانیم اطمینان حاصل کنیم که تنها دریافت کننده صحیح توانایی مشاهده داده‌ها را خواهد داشت.

باید به این نکته دقت داشته باشید که این سطح از حفظ محترمانگی صرفاً برای جابجایی داده‌ها کاربرد دارد. یعنی داده‌های خام در اختیار مالک اطلاعات است و هنگامی که اقدام به ارسال می‌کند TLS داده‌ها را رمزگذاری کرده و در بستری امن منتقل می‌کند، درست در لحظه‌ای که انتقال به پایان می‌رسد و داده‌ها به مقصد می‌رسند، مجدد از حالت رمزگاری خارج شده و به صورت کاملاً خام و واضح در اختیار دریافت‌کننده قرار می‌گیرند.

داده‌ها در صورتی که نیاز داشته باشیم در مقصد نیز باید مجدد رمزگذاری شوند. یعنی هنگامی که اطلاعات را روی دیسک‌های سخت ذخیره می‌کنیم، سایر سیستم‌هایی که به فضای دیسک‌ها دسترسی دارند

نباید بتوانند خارج از فرایند میکروسرویس‌های ما به اطلاعات دسترسی داشته باشند و اگر داده‌ای نیاز دارند باید از طریق نرم‌افزار درخواست دسترسی به آن داده‌ها را ارسال کنند. اغلب نرم‌افزارهای پایگاه داده این امکان را به صورت توکار پیاده‌سازی کرده‌اند و برای داده‌های حساس خود می‌توانیم از این امکانات توکار استفاده کنیم.

۳_۵ دسترسی پذیری^{۱۸} ویژگی مهم نرم‌افزار

یکی از نکاتی که هنگام طراحی سیستم‌های نرم‌افزاری باید به آن توجه کرد این است که سیستم باید دائماً در دسترس باشد. از دسترس خارج شدن یک سیستم نرم‌افزاری می‌تواند ضرر هنگفتی را به کسب و کار وارد کند. در ماه مارچ سال ۲۰۱۶ وبسایت آمازون^{۱۹} برای ۲۰ دقیقه از دسترس خارج شد و ضرر این سایت برای این ۲۰ دقیقه چیزی در حدود ۳/۷۵ میلیون دلار برآورد می‌شود. در تابستان سال ۱۳۹۹ نرم‌افزار سفارشات یک کارگزار بورس ایرانی برای ۱ روز کاری از دسترس خارج شد و ضرری چند ده میلیارد تومانی برای این کارگزاری و سهامداران عضو آن برآورد گردید. دسترسی پذیری یک سامانه نرم‌افزاری صرفاً یک ویژگی و کارکرد امنیتی نیست، بلکه مبحثی عمومی و مربوط به معماری کل سیستم می‌شود. برای مثلاً یک خطای بحرانی در قلب سورس کدهای کسب و کار ممکن است منجر به از دسترس خارج شدن آن شود. اما در بین ویژگی‌های مختلفی که موجود دسترسی پذیری سیستم می‌شود، رعایت نکات امنیتی از اهمیت ویژه‌ای برخوردار است.

در یک سیستم که زیرساخت‌های امنیتی خوبی نداشته باشند، حمله کنندگان ممکن است با حملات DDoS^{۲۰} یا DDoS^{۲۱} تلاش کنند سیستم را برای ارائه خدمات ناتوان کنند. مقابله با این حملات در سطوح مختلف قابل انجام است. در سطح نرمافزار بهترین راه مقابله با این حملات این است که به محض یافتن درخواست‌های نامشروع آن‌ها را از چرخه پردازش خارج کنید. طراحی یک نرمافزار به صورت چند لایه این امکان را فراهم می‌کند که در هر لایه مراقب برخی تهدیدات امنیتی بود و با یافتن آن‌ها، از ایجاد خسارت توسط آن تهدید جلوگیری کنیم. در یک نرمافزار میکروسرویس همه درخواست‌ها از طریق یک دروازه^{۲۲} به API^{۲۳}‌ها می‌رسند. اگر بخواهیم در مورد حملات DDoS تصمیم گیری کنیم بهترین محل جلوگیری از آن‌ها ابتدا در لایه شبکه است. اگر از این لایه عبور کرد بهترین راهکار استفاده از دیواره آتش^{۲۴} است. بعد از آن درخواست‌ها به API Gateway می‌رسد که آن‌جا نیز می‌توانیم با بررسی درخواست‌ها، موارد غیرقانونی را از چرخه حیات خارج کنیم.

۶_۳_ احراز مجوز^{۲۵} و تعیین محدوده اختیارات

احراز هویت به ما می‌گوید که چه فردی با سیستم می‌خواهد کار کند و احراز مجوز تعیین می‌کند که کاربر در چه محدوده‌ای از نرمافزار

20-Denial of service

21-Distributed denial of service

22-API Gateway

23-Application Programming Interface

24-Firewall

25-Authorization

می‌تواند کار خود را انجام دهد. برای مثال در یک سیستم فروشگاه اینترنتی ما ابتدا با احراز هویت مشتری را شناسایی می‌کنیم و پس از آن با احراز مجوز تعیین می‌کنیم که مشتری به چه قسمت‌هایی از نرم‌افزار دسترسی دارد. برای مثلاً امکان ثبت سفارش دارد، می‌تواند کالاها را مورد بررسی قرار دهد و می‌تواند سفارشات پیشین خود را مشاهده کند، اما نمی‌تواند قیمت یک کالا را تغییر دهد یا کالای جدید ثبت کند. در یک سیستم توسعه داده شده بر مبنای میکروسرویس این احراز مجوز معمولاً در مرزهای بیرونی^{۲۶} که همان API Gateway است اتفاق می‌افتد. در اصل ارتباط اگر از بیرون سیستم باشد در سطح مرز و API احراز مجوز می‌شود. ولی ممکن است در هنگام ارتباط داخلی Gateway بین سرویس‌ها نیز نیاز به احراز هویت و مجوز داشته باشیم که در این سطح دیگر داخل سرویس‌ها این عملیات انجام می‌شود.

۴_ امنیت مرزهای^{۲۷} نرم‌افزار

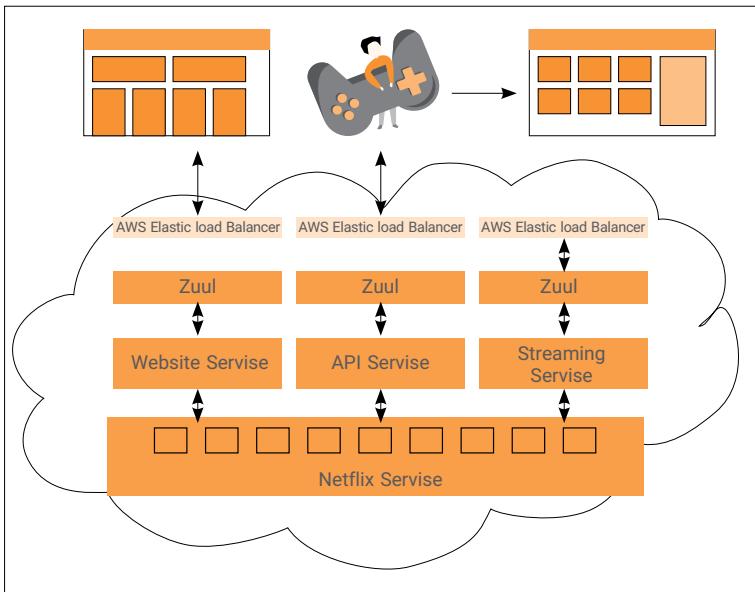
میکروسرویس‌ها هنگام انتشار^{۲۸} معمولاً بطور مستقیم در معرض نرم‌افزارهای مشتری قرار نمی‌گیرند. به طور معمول میکروسرویس‌ها از مجموعه‌ای از API‌ها تشکیل شده‌اند که به واسطه API Gateway در معرض دید جهان خارج قرار می‌گیرند، در عمل با استفاده از API Gateway که در ورودی میکروسرویس‌ها مستقر هستند، تمام پیغام‌های ورودی غربالگری می‌شوند.

26-Edge

27-Edge

28-Deployment

در تصویر صفحه بعد، استقرار میکروسرویس‌های شبیه به میکروسرویس‌های نت فیلیکس^{۲۹} نمایش داده شده است که از API Gate way به نام Zuul استفاده می‌کنند. Zuul امکاناتی نظیر آدرس دهی پویا، رصد کردن، انعطاف پذیری، امنیت و... را فراهم می‌کند. در اصل به عنوان درب ورودی زیرساخت سرور نت فیلیکس عمل می‌کند و ترافیک کاربران نت فیلیکس از سرتاسر جهان را مدیریت می‌کند. در این تصویر، Zuul برای به معرض گذاشتن میکروسرویس Order Processing بواسطه Inventory و استفاده شده است. باقی میکروسرویس‌های مستقر (یعنی Delivery) نیازی به معرض دید قرار گرفتن بواسطه API را ندارند، چرا که نیازی نیست توسط اپلیکیشن‌های بیرونی مورد استفاده قرار بگیرند. در یک استقرار معمول میکروسرویسی، مجموعه‌ای از میکروسرویس‌ها هستند که اپلیکیشن‌های بیرونی نیاز به دسترسی به آنها دارند، و مجموعه‌ای از میکروسرویس‌ها هستند که اپلیکیشن‌های بیرونی نیازی به دسترسی به آنها ندارند. فقط دسته‌ی اول به واسطه‌ی API Gateway به دنیای بیرون عرضه می‌شوند.



مرز ارتباطی میکروسرویس‌ها با دنیای بیرون

۱_F_ نقش API Gateway در دنیای میکروسرویس‌ها

با گذرزمان، API‌ها تبدیل به چهره عمومی شرکت‌ها شدند. اگر این را نکرد، هایم اگر بگوییم یک شرکت بدون API مثل یک رایانه بدون اینترنت است. اگر شما توسعه‌دهنده باشید حتماً درک می‌کنید که زندگی بدون اینترنت به چه شکلی است!

برای بسیاری از شرکت‌ها، API‌ها کانال اصلی درآمدزایی هم هستند. بعنوان مثال در شرکت ایکس‌پدیا^{۳۰} ۹۰ درصد درآمد کل شرکت از طریق API‌ها تامین می‌شود. در سلفورث^{۳۱} ۵۰ درصد و در ای‌بی^{۳۲} ۶۰ درصد درآمد

30-Expedia

31-Saleforce

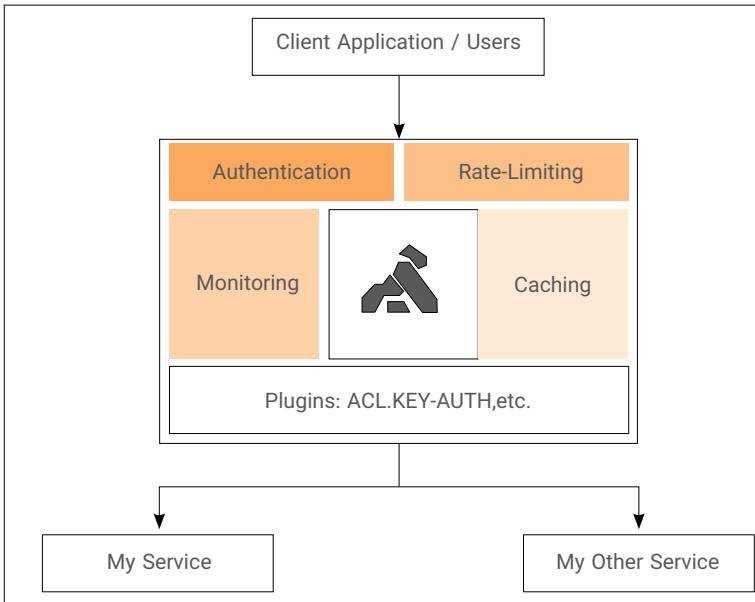
32-EBay

را حاصل می‌کنند. نتفیلیکس هم شرکت دیگری است که بطور جدی روی API‌ها سرمایه‌گذاری کرده است. حساب‌های کاربری نتفیلیکس که درصد قابل توجهی از تمام ترافیک اینترنت شمال آمریکا و جهان را بخود اختصاص داده است، تماماً از طریق API‌های نتفیلیکس منتقل می‌شوند. API و میکروسرویس، دست در دست هم هستند. خیلی از اوقات، یک میکروسرویس که باید برای نرم‌افزار مشتری در دسترس باشد، به عنوان یک API و توسط API عرضه می‌شود. نقش کلیدی API Gateway در استقرار میکروسرویس‌ها عرضه مجموعه‌ای منتخب از میکروسرویس‌ها به دنیای بیرون به عنوان API است و ساخت امکان کیفیت خدمت^{۳۳} (QoS). امکانات QoS شامل امنیت، گلوگاه بودن و تحلیل است. عرضه میکروسرویس‌ها به دنیای بیرون الزاماً به این معنا نیست که در سطح اینترنت بصورت عمومی دیده شوند. ممکن است صرفاً به خارج از دپارتمان‌تان عرضه کنید، یعنی اجازه دهید کاربران یا سیستم‌های یک دپارتمان دیگر درون همان مرزبندی سازمان، با میکروسرویس‌ها بواسطه API Gateway صحبت کنند.

۴_۲ احرازهای در مرزها

مشابه میکروسرویس‌ها، برای API‌ها نیز، مخاطب سیستمی است که بجای خودش یا بجای کاربر انسانی یا سیستم دیگر عمل با API کار می‌کند (تصویر صفحه بعد را مشاهده کنید). اگر چه این امکان فراهم است که کاربر انسانی مستقیماً با API‌ها کار کند اما معمولاً این اتفاق، در غالب اوقات API Gateway با سیستم‌ها کار می‌کند. در ادامه، ما در

مورد کزینه‌هایی که برای احراز هویت یک سیستم (یا نرم‌افزار مشتری) در API Gateway داریم صحبت خواهیم کرد.



احراز هویت در مرزهای ورودی توسط API Gateway

احراز هویت مبتنی بر گواهینامه از API در مرز با استفاده از MTLS^{۳۴} محافظت می‌کند. در استقرار میکروسرویس‌های نتفیلیکس، دسترسی به API‌ها با استفاده از گواهینامه‌ها محافظت می‌شود. فقط مشتریانی که گواهینامه‌ی معتبر دارند امکان دسترسی به API‌های نتفیلیکس را دارند. نقش API Gateway در اینجا این است که اطمینان^{۳۵} حاصل کند که فقط مشتریان دارای گواهینامه‌ی قابل اطمینان به API دسترسی دارند و

33-Quality of service

34-Mutual Transport Layer Security

35-Trusted certificate

فقط درخواست‌های این مشتریان است که به سمت میکروسرویس‌های سرویس دهنده^{۳۶} فرستاده می‌شوند.

هر کسی می‌تواند برنامه‌ای بسازد که از API‌های فیسبوک^{۳۷} و توییتر^{۳۸} استفاده^{۳۹} کند. برنامه‌ها هم می‌توانند نرم‌افزارهای تحت وب یا موبایل باشند. نرم‌افزار به API می‌تواند بجای خود یا کاربر انسانی دسترسی داشته باشد. OAuth 2.0 یک چارچوب^{۴۰} احراز صلاحیت برای نمایندگی کنترل دسترسی است. این روش، برای محافظت از API‌ها در زمانی است که یک سیستم تقاضای دسترسی به API بجای یک سیستم دیگر یا کاربر را دارد توصیه شده است.

نقش API Gateway اعتبار سنجی نشانه‌های^{۴۱} امنیتی 2.0 OAuth می‌باشد که در هر درخواست API وجود دارد. نشانه امنیتی OAuth شامل هم برنامه‌ی ثالث می‌شود هم کاربری که وکالت دسترسی به برنامه‌ی ثالث جهت دسترسی به API از طرف خودش را داده است. یکی دیگر از مواردی که در مرزها و در API Gateway قابل بررسی است، احراز مجوز است. با توجه به مرکزیت این قسمت، قبل از انجام کار، مجوزهای کاربر بررسی می‌شود که شامل دسترسی به منبع مورد نظر باشد و در صورتی که مجوز این کار را دارا نبود جلوی دسترسی گرفته می‌شود. تمام اتصالات مشتری‌ها به سرویس‌ها به دروازه API ختم می‌شود.

36-Upstream microservice

37-Facebook

38-Twitter

39-Consume

40-Framework

41-Token

اگر همه شرایط مساعد باشد، درخواست‌ها به میکروسرویس‌های اصلی سرویس‌دهنده منتقل می‌شوند. راهی نیاز است تا از کانال‌های ارتباطی بین دروازه و میکروسرویس مربوطه محافظت شود، البته این محافظت باقیستی برای ارسال اطلاعات^{۴۲} بستر اولیه کاربر/مشتری نیز لحاظ گردد.. اطلاعات بستر اولیه کاربر شامل یکسری اطلاعات پایه‌ای درمورد کاربر نهایی و همین‌طور اطلاعاتی درمورد نرمافزار مشتری می‌باشد. این اطلاعات عموماً مورد استفاده میکروسرویس سرویس‌دهنده برای کنترل دسترسی در سطح سرویس قرار می‌گیرد.

همانطور که به احتمال زیاد حس می‌زنید، ارتباط بین API Gateway و میکروسرویس‌ها از نوع سیستم به سیستم است، پس می‌توانید از احراز هویت MLTS برای امن کردن کanal استفاده کنید. اما چطور باید مفاد کاربر را برای میکروسرویس سرویس‌دهنده ارسال کنید؟ چند گزینه دارید: ارسال مفاد کاربر در سربرگ HTTP یا ایجاد JWT^{۴۳} با داده‌های کاربر. گزینه‌ی اول سر راست است، اما کمی گرفتاری در اطمینان دارد: زمانی که اولین میکروسرویس همان مفاد کاربر را در سربرگ HTTP برای یک میکروسرویس دیگر ارسال می‌کند، میکروسرویس دوم هیچ تضمینی مبنی بر اینکه مفاد کاربر تغییر نکرده‌است ندارد. اما با جL شما ماهیتی دارید که با واسطه‌گری بین ارتباط^{۴۴} امکان تغییر بدون شناسایی آن نیست، چرا که صادر کننده‌ی جL آن را امضا کرده است. اگر با جL آشنایی ندارید آن را محتوای امضا شده‌ای تصور کنید که داده

42-Initial context

43-Json web token

44-Man in the middle 45-Domain

(در این مثال مفاد کاربر) را در حالت رمز شده و مطمئن منتقل می‌کند. دروازه می‌توانند JWT‌ای شامل مفاد کاربر (و یا مفاد مشتری) را ایجاد کنند و آن را برای میکروسرویس سرویس دهنده ارسال کنند. میکروسرویس دریافت کننده با تایید کردن امضای JWT به کمک کلید عمومی بخشی که JWT را صادر کرده است، آن را اعتبارسنجی می‌کند.

۵. امن کردن ارتباط سرویس به سرویس

تعدد ارتباطات سرویس با سرویس در استقرار میکروسرویس‌ها بالاست. ارتباطات بین دو میکروسرویس می‌تواند درون یک حوزه^{۴۶} مطمئن یا بین دو حوزه مطمئن باشد. حوزه مطمئن بیانگر مالکیت است. میکروسرویس‌ها با هم و احتمالاً درون یک حوزه مطمئن یا یک مرزبندی مطمئن که در سطح سازمان تعریف شده است و در نظر گرفتن فاکتورهای دیگر، توسعه، نصب و مدیریت می‌شوند.

برای مدل امنیتی که به منظور محافظت از ارتباطات سرویس به سرویس توسعه می‌دهید، باید با در نظر گرفتن کانال‌های ارتباطی بین مرزبندی‌های مطمئن باشد و همینطور چگونگی ارتباطاتی که در واقع می‌خواهیم بین میکروسرویس‌ها باشد: همگام^{۴۷} یا ناهمگام^{۴۸}. غالب اوقات ارتباطات همگام بر بستر HTTP رخ می‌دهد. ارتباطات ناهمگام می‌تواند با هر نوع سیستم Amazon پیغام‌رسانی^{۴۹} نظیر RabbitMQ، Kafka، ActiveMQ و یا حتی Simple Queue Service (SQS) رخ دهد.

46-Synchronously

47-Asynchronously

48-Messaging

۱_۵_ احراز هویت سرویس به سرویس

برای امن کردن ارتباطات سرویس‌ها در قالب میکروسرویس سه روش معمول وجود دارد: اطمینان به شبکه^{۴۹}، MTLS^{۵۰} و WTLS‌ها.

روش اطمینان به شبکه قدیمی‌ترین راه حل است که در عمل امنیتی را هم در ارتباطات سرویس به سرویس اجبار نمی‌کند. در واقع این مدل به امنیت در لایه شبکه اکتفا می‌کند. امنیت لایه شبکه باید این تضمین را بدهد که هیچ حمله‌کننده‌ای امکان شنود ارتباطات میکروسرویس‌ها را ندارد. همچنین هر میکروسرویس بعنوان سیستم مطمئن شناخته می‌شود. این انتخاب می‌تواند مبنی بر انتظار شما از سطح امنیت و اعتماد شما به اجزای درون شبکه باشد.

یک روش قدیمی دیگر، با عنوان شبکه‌ی نامطمئن^{۵۱} شناخته می‌شود که برخلاف روش اطمینان به شبکه است. این روش این پیش‌فرض را دارد که شبکه همواره نامطمئن و آسیب‌پذیر است و به هیچ جزئی دسترسی‌ای اعطا نمی‌شود. هر درخواستی که به هر گره می‌رسد، پیش از آنکه برای پردازش پذیرفته شود، هریار باید احراز هویت و احراز مجوز شود.

یکی از روش‌های مرسوم دیگر برای امن کردن ارتباطات سرویس به سرویس در استقرار میکروسرویس‌ها، روش Mutual TLS می‌باشد. در حقیقت، این روش پر استفاده‌ترین شکل از احراز هویت در این روزهای است. هر میکروسرویسی که در استقرار وجود دارد، برای ارتباط با میکروسرویس گیرنده باید جفت کلیدهای عمومی/خصوصی را با خود

حمل کند و با استفاده از MTLS احراز هویت کند.

TLS در انتقال داده، اطمینان خاطر و یکپارچگی را به ارمغان می‌آورد و کمک می‌کند مشتری سرویس را تشخیص دهد. میکروسرویس مشتری می‌فهمد که کدام میکروسرویس می‌خواهد با او صحبت کند. اما با TLS (نوع یک طرفه) امکان تشخیص میکروسرویس مشتری برای میکروسرویس گیرنده نیست. در اینجاست که MTLS وارد می‌شود (Mutual=Mتقابل). با TLS هر میکروسرویس در ارتباطات می‌تواند طرف مقابل خود را شناسایی کند.

سومین روش برای امن کردن ارتباطات سرویس به سرویس در استقرار میکروسرویس‌ها، روش JWT است. برخلاف TLS، JWT در لایه‌ی کاربرد کار می‌کند، نه لایه‌ی انتقال^{۵۱} JWT در عمل همچون یک ظرف عمل می‌کند که می‌تواند مجموعه‌ای از چیزها را از جایی به جای دیگر منتقل کند.

این چیزها، هرچیزی می‌تواند باشد، مثل خصوصیات کاربر نهایی (آدرس ایمیل، شماره تلفن)، حقوق کاربر نهایی (این که چه کارهایی می‌تواند انجام دهد) یا هر چیز دیگری که میکروسرویس فراخوانی شده می‌خواهد برای میکروسرویس گیرنده ارسال کند. JWT شامل این چیزها می‌شود و آن‌ها را هم با صادرکننده JWT امضا می‌کند. صادرکننده می‌تواند یک خارجی باشد یا خود میکروسرویس فراخوانی شده باشد.

هر میکروسرویس باید جفت کلید خودش را داشته باشد و برای امضای JWT از کلید خصوصی استفاده کند. در اکثر مواقع احراز هویت مبتنی بر JWT بر بستر TLS کار می‌کند: JWT احراز هویت را انجام می‌دهد و TLS قابلیت اطمینان و یکپارچگی در انتقال داده را به عهده می‌گیرد.

^{۵۱}- اشاره به لایه‌های پشت‌هه شبکه Application و لایه Transport

۵_۲ احراز مجوز در سطح سرویس

در یک استقرار معمول میکروسرویس‌ها، احراز صلاحیت می‌تواند در مرز صورت بگیرد یا در هر سرویس یا در هر دو جا. وقتی احراز صلاحیت در سطح سرویس انجام شود، دست بازتری برای اعمال سیاست‌های کنترل دسترسی‌هایی که دلمان می‌خواهد داشته باشیم، خواهیم داشت. دو روش برای اعمال صلاحیت دسترسی در سطح سرویس مطرح است:

مدل مرکزی PDP^{۵۲} و مدل توکار PDP.

در مدل مرکزی PDP، تمام سیاست‌های کنترل دسترسی، در یک جای مرکزی، ایجاد، نگهداری و اجرا می‌شوند (تصویر صفحه بعد را مشاهده کنید). هر باری که سرویس بخواهد یک درخواست را اعتبارسنجی کند، باید با نقطه انتهایی^{۵۳} که توسط PDP مرکزی عرضه شده‌است صحبت کند. این روش به طور جدی به PDP وابسته است و تاخیر را به علت هزینه فراخوانی از راه دور با نقطه انتهایی PDP افزایش می‌دهد. در این موارد، با ذخیره موقت^{۵۴} سیاست‌های کنترل دسترسی در سطح سرویس می‌توانیم این تاخیر را از بین ببریم، اما وقتی این ذخیره موقت بخارط محدودیت زمانی از بین برود، یا برای دریافت سیاست‌های جدید، طبعاً دوباره باید با PDP ارتباط بگیرد. در عمل، سیاست‌های جدید به ندرت اضافه می‌شوند، اما انقضای زمانی حافظه‌ی موقت مسئله‌ای پر رخداد است.

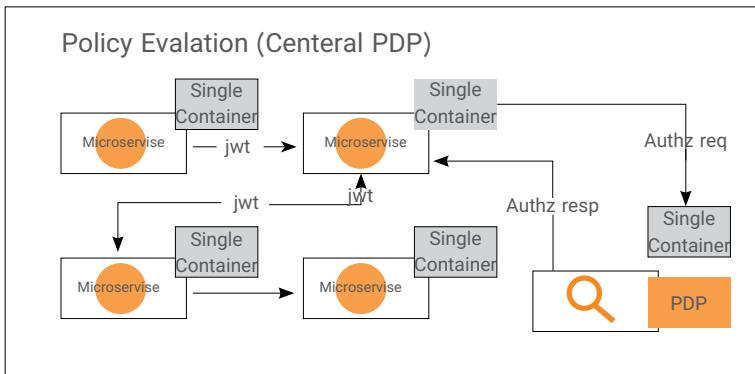
با PDP توکار، سیاست‌ها به شکل مرکزی تعریف، اما در سطح سرویس

52-Policy decision point

53-Endpoint

54-Cache

نگهداری و اجرا می شوند. چالش PDP‌های توکار این است که چطور سیاست‌ها را از طریق PAP^{۵۵} مرکزی بروزرسانی کنیم. برای انجام این کار دو روش متقابل وجود دارد. یک روش دریافت مستمر از PAP بعد از یک بازه زمانی می‌باشد و مجدد دریافت سیاست‌های جدید از PAP. روش دیگر، مکانیزم فرستادن^{۵۶} است. زمانی که سیاستی تغییر کرد یا اضافه شد، خود PAP آن را بعنوان یک رخداد منتشر می‌کند. در این حالت، هر میکروسرویس بعنوان دریافت کننده رخداد^{۵۷} عمل می‌کند و در سیاست‌های مربوط به خودش را از PAP می‌گیرد و PDP توکار خودش را بروز می‌کند.



احراز مجوزها به صورت مرکزی

برخی از افراد معتقد هستند که هر دوی این روش‌ها مخرب هستند. آن‌ها سیاست‌ها را فقط در زمانی که سرور بالا می‌آید روی PDP توکار

55-Policy Administration Point

56-Pull

57-Push

58-Event consumer

بارگذاری می‌کنند. و هر زمانی که سیاست جدیدی اضافه شد یا تغییر کرد، تمام سرویس‌ها مجبور به شروع مجدد^{۵۹} هستند.

۳_۵_ انتشار اطلاعات زمینه‌ای کاربر

زمانی که یک میکروسرویس یک میکروسرویس دیگر را فراخوانی می‌کند، نیاز دارد که شناسه‌ی کاربرنها بی‌ی و شناسه‌ی خود میکروسرویس را حمل کند. زمانی که یک میکروسرویس با استفاده از JWT و MTLS احراز هویت می‌کند، شناسه‌ی میکروسرویس فراخوانی شده درون اعتبارنامه‌ی توکار آن تزریق می‌شود. سه راه معمول برای ارسال مفاد کاربرنها بی‌ی از یک میکروسرویس به یک میکروسرویس دیگر وجود دارد: روش اول ارسال مفاد کاربر در سربگ HTTP است. این تکنیک به میکروسرویس دریافت کننده کمک می‌کند تا کاربر را تشخیص دهد، اما لازمه این کار اعتماد دریافت کننده به فراخوانی کننده است. اگر فراخوانی کننده بخواهد که دریافت کننده را فریب دهد به آسانی فقط با تنظیم نام دلخواهش در سربگ HTTP می‌تواند اینکار را انجام دهد. روش دوم استفاده از JWT است. JWT اطلاعات زمینه‌ای کاربر را از میکروسرویس فراخوانی کننده به میکروسرویس گیرنده حمل می‌کند و او هم آن را درون سربگ HTTP قرار می‌دهد. اگر قرار باشد JWT خودش صادرکننده باشد، هیچ ارزش افزوده‌ای نسبت به روش اول ندارد. JWT که خودش صادرکننده است، توسط خود سرویس فراخوانی کننده امضا می‌شود، فلذا براحتی می‌تواند میکروسرویس دریافت کننده را با اضافه کردن هر نام دلخواهی که می‌خواهد فریب دهد.

سومین روش نیز استفاده از JWT‌ای که توسط یک STS خارجی صادر شده‌است و برای تمام ماشین‌های درون استقرار قابل اطمینان است. اطلاعات زمینه‌ای کاربر که درون JWT قرار گرفته است امکان تغییر ندارد، چنانچه تغییر کند، امضای JWT نامعتبر می‌شود. این امن‌ترین روش است. زمانی که STS خارجی دارد، میکروسرویس فراخوانی کننده می‌تواند آن را درون یک JWT جدید دیگر به کار بگیرد. اینکار JWT تو درتو ایجاد می‌کند.

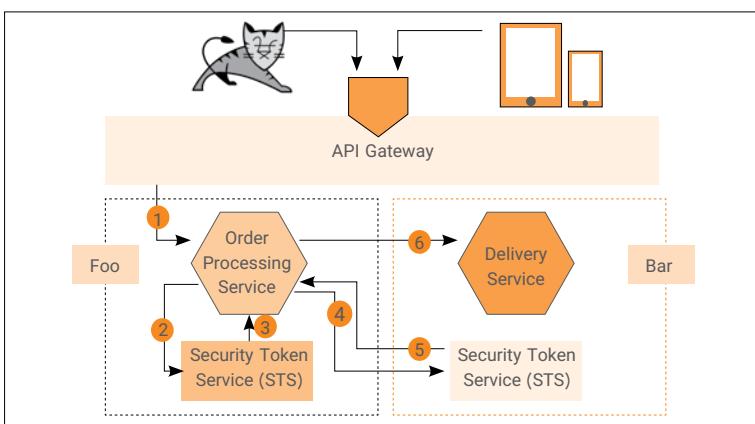
۴_۵_ عبور از محدوده امن

به طور معمول هنگام انتشار میکروسرویس‌ها چندین محدوده مورد اطمینان خواهیم داشت. این محدوده‌های مورد اطمینان آن‌هایی هستند که تیم‌های توسعه میکروسرویس بر روی استراتژی‌ها و نحوه مدیریت آن‌ها نظارت دارند یا به کمک روابط سازمانی می‌توانند آن‌ها را کنترل کنند. برای مثلاً در یک سیستم خرده فروشی، واحد خرید و فروش احتمالاً بر روی تمامی سرویس‌های توسعه داده شده در این حوزه کنترل و نظارت دارد و محدوده مورد اطمینان خود را می‌سازد.

هنگامی که دو سرویس در یک محدوده اطمینان با هم صحبت می‌کنند، آن سرویس‌ها به STS مرکزی اطمینان دارند و که با آن‌ها در یک دامنه اطمینان قرار دارد. با توجه به این اطمینان موجود بین سرویس‌ها و STS، میکروسرویس‌ها می‌توانند نشانه‌های امنیتی که برای آن‌ها ارسال می‌شود را به سادگی بررسی و صحبت سنجری کنند. معمولاً در یک دامنه مورد اطمینان یک STS هم وجود دارد که همه

میکروسرویس‌های حاضر در این دامنه از آن برای صحت سنجی و انتشار نشانه‌های امنیتی استفاده می‌کنند.

اگر یک میکروسرویس بخواهد با میکروسرویس خارج از محدوده اطمینان جاری خود ارتباط برقرار کند دو حالت کلی برای انجام این کار وجود دارد. برای تشریح اولین حالت تصویر صفحه بعد را مشاهده کنید. فرض کنید که سرویس Order Processing در دامنه اطمینان Foo می‌خواهد با میکروسرویس Delivery Service در دامنه اطمینان Bar ارتباط برقرار کند. ابتدا میکروسرویس Order Processing باید یک نشانه امنیتی که مورد قبول میکروسرویس‌های حاضر در محدوده Bar هستند را به دست آورد. به بیان دیگر سرویس حاضر در محدوده Foo باید برای دریافت نشان امنیتی از طریق STS حاضر در محدوده امنیتی Bar اقدام نماید. مراحل انجام این کار به شرح زیر می‌باشد.



ارتباط بین دو دامنه مورد اطمینان پشت یک API Gateway

■ در اولین قدم API Gateway درخواست را از اپلیکیشن مشتری به میکروسرویس Order Processing در حوزه مطمئن Foo بهمراه JWT امضا شده توسط API یا توسط STS متصل به آن ارسال می‌کند. از آنجایی‌که تمام میکروسرویس‌های حوزه مطمئن Foo به STS بالاترین سطح اطمینان دارند، میکروسرویس Order Processing نشانه امنیتی را به عنوان نشانه معتبر می‌پذیرد. JWT خصیصه‌ای دارد بنام Aud که سیستم هدف JWT را مشخص می‌کند.. در این مورد، مقدار Aud با میکروسرویس که میکروسرویس JWT حوزه Foo تنظیم شده است. در حالت ایده‌آل، زمانی دریافت کند، باید آن JWT را نادیده بگیرد، حتی اگر امضای آن معتبر باشد.

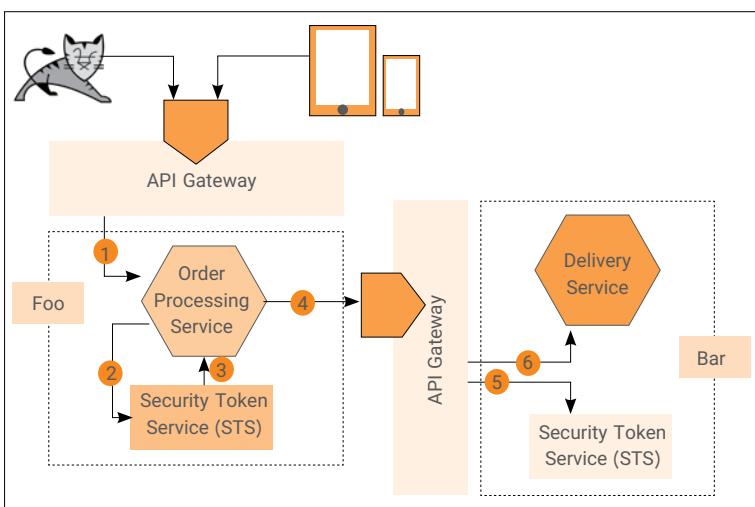
■ قدم دوم این است که میکروسرویس Order Processing، JWT که از API Gateway یا سطح بالاتر گرفته است برای STS درون حوزه مطمئن Foo ارسال می‌کند. تکرار می‌کنم، STS حوزه Foo باید مقدار Aud درون JWT ای که دریافت کرده است را اعتبارسنجی کند. اگر نتواند نشانه آن را شناسایی کند، باید آن را نادیده بگیرد.

■ سومین قدم این است که STS درون Foo یک JWT جدید که توسط خودش امضا شده است و مقدار Aud آن را برابر با STS حوزه Mطمئن Bar قرار داده است برمی‌گردداند.

■ در چهارمین و پنجمین قدم میکروسرویس Order Processing به STS حوزه Mطمئن Bar دسترسی دارد و JWT بدست آمده از قدم سوم را با JWT امضا شده توسط STS حوزه Mطمئن Bar جابجا می‌کند و البته مقدار Aud آن را برابر با میکروسرویس Delivery قرار می‌دهد.

قدم ششم میکروسرویس Order Processing به میکروسرویس Delivery بواسطه JWT بدست آمده از قدم پنج دسترسی دارد و از آنجایی که حوزه JWT، Bar را امضا کرده است و مقدار Aud درست می باشد، میکروسرویس Delivery توکن را می پذیرد.

اما در حالت دوم سرویس ها پشت یک API Gateway قرار ندارند. در این حالت هر میکروسرویس در حوزه اطمینان خود و با یک API Gateway جداگانه قرار دارد که نمونه ای از این حالت را در تصویر ۸ مشاهده می کنید.



ارتباط بین دو دامنه مورد اطمینان پشت دو API Gateway مجزا

در چنین شرایطی انجام ارتباط به شرح زیر صورت خواهد پذیرفت.

در اولین قدم API Gateway درون حوزه مطمئن Foo درخواست را از نرم افزار مشتری به سمت میکروسرویس Order Processing بهمراه JWT امضا شده توسط API Gateway یا توسط STS درون Foo که

به API Gateway حوزه چسبیده شده ارسال می‌کند. از آنجایی که تمام میکروسرویس‌های درون حوزه Foo به STS این حوزه اعتماد دارند، میکروسرویس Order Processing توکن رسیده را بعنوان معتبر می‌پذیرد. در قدم دوم میکروسرویس Order Process، JWT اصلی را که از API Gateway یا همان STS حوزه Foo دریافت کرده است را برای STS خود ارسال می‌کند.

قدم سوم به این شکل است که STS حوزه Foo یک JWT جدید برمی‌گرداند که توسط خودش برای API Gateway درون حوزه مطمئن امضا کرده است. Bar

در چهارمین قدم میکروسرویس Order Processing توسط JWT ای ای ای که در قدم سوم بدست آمد، به میکروسرویس Delivery درون حوزه Bar دسترسی پیدا می‌کند. از آنجایی که API Gateway حوزه Bar به STS حوزه Foo اطمینان دارد، JWT را معتبر تشخیص می‌دهد. API Gateway توسط STS حوزه Foo امضا شده است و با حوزه Bar همخوانی دارد.

قدم پنجم این است که API Gateway Bar با STS حوزه Bar تا JWT خودش را ایجاد کند (یعنی توسط STS حوزه Bar امضا شده باشد).

در انتها و ششمین قدم API Gateway Bar حوزه Bar درخواست را به همراه JWT صادر شده از STS حوزه Bar به سمت میکروسرویس Delivery ارسال می‌کند. از آنجایی که میکروسرویس Delivery به STS خودش اطمینان دارد، نشانه بعنوان معتبر پذیرفته می‌شود.

۶_ بحث و ارزیابی و نتیجه‌گیری

در حوزه امنیت میکروسرویس‌ها در دو قسمت جداگانه این امنیت قابلیت بررسی و پیاده‌سازی است. حفظ امنیت در مزهای و حفظ امنیت در داخل هر میکروسرویس. این دو محل را برای پیاده‌سازی ویژگی‌های امنیتی نرم‌افزار می‌توانیم از جنبه‌های مختلفی با هم مقایسه کنیم. در ابتدا حوزه اختیارات این دو قسمت را می‌توانیم با هم مقایسه کنیم، هنگام پیاده‌سازی امنیت نرم‌افزار در مزهای بیرونی، تسلط کامل به کل فرایند نرم‌افزار و همه میکروسرویس‌ها وجود دارد و در صورت نیاز می‌توان یک درخواست را در حوزه چندین میکروسرویس از منظر امنیتی بررسی کرد، اما این تسلط به کل اجزا در سطح هر میکروسرویس وجود ندارد و یک سرویس توانایی این را ندارد که هنگام برقراری مباحث امنیتی شرایط سایر سرویس‌ها را نیز در نظر بگیرید. اگر هم این امکان را فراهم کنیم وابستگی شدیدی بین دو سرویس ایجاد می‌شود و اصول اولیه معماري و محاسن آن از بین می‌رود.

با توجه به انجام مرکزی همه فرایندها هنگام پیاده‌سازی امنیت در مزهای نرم‌افزار، اطمینان خاطر بیشتری می‌توان به کل سیستم داشت، در صورتی که یک بخش جدید به کل سیستم اضافه شود، نگرانی بابت پیاده‌سازی مباحث امنیتی نداریم و یک بار این کار را انجام داده‌ایم، وقتی جزء جدیدی به سیستم اضافه می‌شود به صورت خودکار از مزایا و امکانات امنیتی موجود در سیستم بهره‌مند می‌شود.

با کمی دقت در فرایندهای امنیتی می‌توانیم دریابیم که این جامعیت

و پیاده‌سازی مرکزی همیشه به نفع کل سیستم نیست، در صورتی که حفره امنیتی در فرایند کاری وجود داشته باشد و کشف شود، کل سیستم مورد تهدید قرار می‌گیرد و عملیات کل سیستم تخریب می‌شود، اما اگر این کار به ازای میکروسرویس‌ها انجام شده باشد، در صورتی که ایرادی نیز وجود داشته باشد، احتمالاً فقط یکی از میکروسرویس‌ها تهدید می‌شود و سایر سیستم‌های موجود می‌توانند به کار خود ادامه دهند.

با توجه به اینکه مشاهده کردیم که در مرزهای نرم‌افزار به عنوان API Gateway استفاده می‌شوند تنوع زیادی دارند، اگر کل فرایند امنیت سیستم داخل آن پیاده‌سازی شود، با تغییر تکنولوژی و نیاز به عوض کردن ابزار مورد استفاده، کل فرایندهای طراحی شده برای امنیت نرم‌افزار نیز از بین می‌رود و مجدداً نیاز به طراحی و پیاده‌سازی با ابزار جدید دارد که خود این ایراد ممکن است ما را از تعویض ابزار و استفاده از مورد مناسب‌تر منصرف کند. در صورتی که اگر امنیت را در سطح میکروسرویس‌ها تامین کرده باشیم، به سادگی می‌توانیم ابزارهایی که در مرزها استفاده کرده‌ایم را تغییر دهیم و نگران از دست رفتن یکی از ویژگی‌های حیاتی نرم‌افزارمان نیز نباشیم.

نرم‌افزارها به دلایل مختلفی نیاز دارند که بررسی‌های امنیتی را در حین اجرا انجام دهند، در صورتی که تعداد میکروسرویس‌های ما زیاد باشد، و درخواست‌های زیادی نیز به سامانه نرم‌افزاری ما وارد شود، بار زیادی برای انجام فرایندهای امنیتی در مرزها ایجاد می‌شود و تعداد رفت و آمد‌های بین میکروسرویس‌ها و ابزارهای امنیتی موجود در مرزها بسیار زیاد می‌شود و ممکن است باعث ایجاد گلوگاه شده و کل کارکرد سیستم

مختل شود. در عوض اگر این کار در میکروسرویس‌ها انجام شود، هم درخواست به یک بخش خاص بیش از حد توان آن نمی‌شود و هم تعداد درخواست‌ها و پاسخ‌های امنیتی در سیستم کاهش می‌یابد و در هنگام بار زیاد سیستم دچار اختلال نمی‌شود.

در صورتی که بخواهیم امنیت را در مرزها برقرار کنیم، با توجه به ماهیت میکروسرویس‌ها باید داده‌های زیادی از هر میکروسرویس در مرزها وجود داشته باشد تا ابزارهای امنیتی موجود در مرزها بتوانند تمامی نیازهای امنیتی میکروسرویس‌ها را برآورده کنند که با این اتفاق، هم وابستگی شدیدی بین سرویس‌ها و ابزارهای موجود در مرزها اتفاق می‌افتد هم نیاز است که به ازای اضافه شدن هر بخش جدید، تغییراتی در مرزها نیز رخ دهد و تنظیمات امنیتی میکروسرویس جدید هم در مرز انجام شود که این اتفاق اگر با خطأ همراه باشد، کل فرایند امنیتی سیستم را دچار اختلال می‌کند، اما پیاده‌سازی امنیت در خود میکروسرویس‌ها این ایراد را ندارد. هر سرویس دقیقاً به ویژگی‌ها و نیازمندی‌های خود آگاهی دارد و می‌تواند فرایندهای مورد نیاز خود را طراحی و پیاده‌سازی نماید.

جدای از سطح اطلاع مرزها از عملیات داخلی میکروسرویس‌ها، از قبل می‌دانیم که همه میکروسرویس‌ها نیازی به قرار گرفتن پشت مرزها را ندارد و اگر فرایند امنیتی در مرزها اتفاق بیوافتد، حتی سرویس‌هایی که خدماتی از بیرون از مرز دریافت نمی‌کنند یا کارکردی برای خارج از مرزها ارائه نمی‌دهند هم باید به ابزارهای موجود در مرزها وابسته شوند که ایجاد این وابستگی بیهوده به نظر می‌رسد.

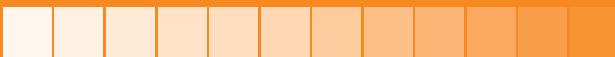
با تمام این توضیحات، هر کدام از این روش‌ها مزايا و معایب دارند که انتخاب بین يکی از این روش‌ها را بسیار سخت می‌کند. با کمی دقت در مزايا و معایب اين روش‌ها در میابیم که اين دو روش، نباید جدائی از يکديگر در نظر گرفته شوند که هنگام طراحی سیستم بخواهیم يکی از اين روش‌ها را برای برقراری امنیت سیستم انتخاب کنیم. بلکه با طراحی درست سیستم، می‌توانیم از هر دو روش در امتداد يکديگر استفاده کنیم و در لایه‌های مختلفی از نرم‌افزار امنیت را به شکل صحیحی و با ابزار درست برقرار کنیم. در مرزهای سیستم صرفا بررسی‌ها و چک‌های اولیه انجام شود و اطلاعات اولیه استخراج شده و در صورت وجود حمله و خرابکاری اجازه ورود درخواست به سیستم داده نشود، و پیاده‌سازی جزئیات امنیتی هر میکروسرویس مانند احراز مجوزها و دسترسی دادن به بخش‌های خاص و ... داخل هر میکروسرویس به طور مجزا انجام شود. در این حالت نیز برای جلوگیری از نیاز به پیاده‌سازی کدهای تکراری می‌توان قابلیت‌های مورد نیاز را در قالب یک کتابخانه ایجاد کرد که هر میکروسرویس به طور مجزا از آن استفاده می‌کند.

سنجهش میزان برتری در کدنویسی با تعداد خطوط کد مثل سنجهش

میزان وزن در حین ساخت هوایپیماست؛ پس هرچه کمتر بهتر.

بیل گیتس

فصل هشتم: آشنایی با روش‌های انتشار



- نصب چند سرویس مختلف به ازای هر میزبان
- نصب یک سرویس به ازای هر میزبان
- نصب و راهاندازی به کمک کانتینرها

مقدمه:

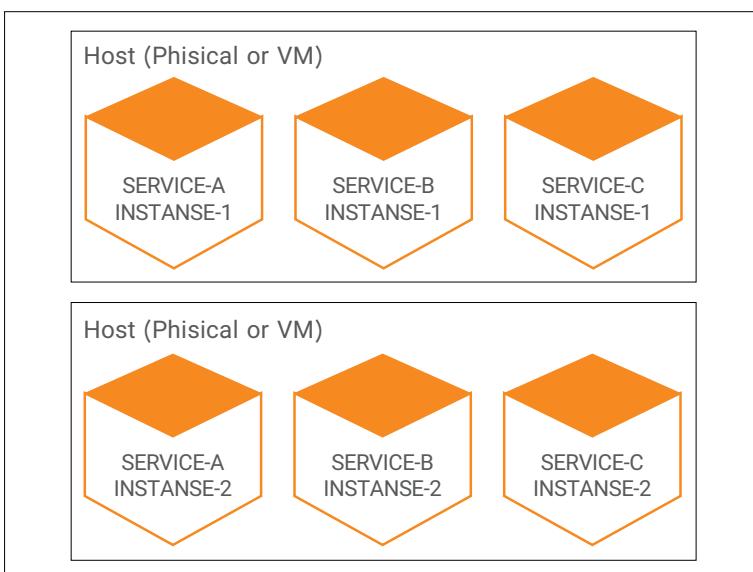
اگر از معماری Monolithic برای توسعه سرویس‌های خود استفاده کنیم، در انتهای نیاز داریم برنامه‌ای را روی یک سرور نصب کنیم. یا اگر برنامه ما قرار باشد بار زیادی را تحمل کند، چندین سرور تهیه می‌کنیم و نرمافزار خود را روی چند سرور نصب می‌کنیم و به کمک یک Load Balancer بار را روی نمونه‌های مختلف توزیع می‌کنیم. هرچند همیشه کار به همین سادگی نیست اما در پیچیده‌ترین حالات هم معمولاً توزیع نرمافزارهای Monolithic ساده‌تر از میکروسرویس‌ها است.

در مقابل هنگامی که از معماری میکروسرویس استفاده می‌کنیم، برای یک برنامه ممکن است دهها یا صدها سرویس داشته باشیم. هر کدام از این سرویس‌ها ممکن است با زبان و فریمورکی خاص توسعه داده شده باشند. هر کدام از این میکروپلیکیشن‌هایی که تولید می‌کنیم، اندازه‌های متفاوت، بارهای متفاوت، درخواست‌های متفاوت و زیرساخت‌های نگهداری و مانیتورینگ متفاوتی هم ممکن است داشته باشند. ممکن است شما نیاز داشته باشید با توجه به تعداد درخواست‌هایی که برای هر سرویس ارسال می‌شود، تعداد متفاوتی نسخه از این سرویس را اجرا کنید. هر نسخه از این سرویس‌ها هم باید زیرساخت مناسبی از نظر CPU, RAM و ... در اختیار داشته باشند. در کنار همه این پیچیدگی‌ها نصب و راه‌اندازی

نرم‌افزار ما باید سریع، مطمئن و مقرر و به صرفه باشد. روش‌های متفاوتی برای دست‌یابی به این اهداف وجود دارد که در ادامه به بررسی این روش‌ها می‌پردازیم.

■ روش اول: نصب چند سرویس مختلف به ازای هر میزبان

یکی از راه‌های انتشار میکروسرویس‌ها نصب و راهاندازی چندین سرویس روی یک سرور است. این روش که یکی از قدیمی‌ترین روش‌های انتشار نرم‌افزار است چندین سرور فیزیکی یا مجازی تهیه می‌کنیم و سرویس‌های متفاوتی را روی هر کدام از این سرورها نصب می‌کنیم. در این روش معمولاً هر نسخه از سرویس‌های ما روی آی‌پی‌ها و پورت‌هایی کاملاً مشخص و از ابتدا تنظیم شده اجرا می‌شوند. برای آشنایی بهتر با این روش به تصویر زیر دقت کنید.



مانند هر کار دیگری انتخاب این روش مزایا و معایبی دارد که در ادامه با هم بررسی می‌کنیم. استفاده بهینه از منابع و زیرساخت‌ها یکی از بزرگترین مزایای استفاده از این روش است. در این روش چندین برنامه مختلف امکاناتی مانند سخت‌افزار، سیستم‌عامل، وب سرور و ... را با هم به صورت مشترک استفاده می‌کنند.

سرعت بالای نصب و راهاندازی یکی دیگر از مزایای استفاده از این روش است. شما می‌توانید به سادگی نسخه‌ای از نرم‌افزار را روی سرورهای مختلف کپی می‌کنیم و برنامه را اجرا می‌کنیم. اگر انتخاب شما .NET است. باشد به سادگی Assembly‌های خود را روی سرورها کپی می‌کنید. در مورد جاوا هم می‌توانید JAR یا WAR فایل‌های خود را روی سرور هدف کپی کنید و اگر از Node.js استفاده کنید باید سورس برنامه‌های خود را روی سرور کپی کنید.

در کنار این مزایا معایبی را هم می‌توانیم برای استفاده از این روش بیان کنیم. با توجه به اینکه همه سرویس‌های مختلف روی یک ماشین و بدون هیچ محدودیتی نسبت به هم اجرا می‌شوند. در این روش نمی‌توانید به سادگی محدودیتی برای استفاده از منابع سیستم تعیین کنید و یکی از سرویس‌ها می‌تواند در صورت بروز مشکل تمامی منابع سیستم را مصرف کرده و سایر سرویس‌ها هم در این شرایط دچار اختلال می‌شوند.

ایراد دیگری که استفاده از این روش دارد نیاز به دانستن جزئیات نصب و راهاندازی یک سرویس برای تیم زیرساخت است. همانگونه که گفته شد، سرویس‌های مختلف ممکن است از زبان‌های برنامه نویسی و زیرساخت‌های متفاوتی استفاده کنند و در این شرایط تیم زیرساخت و

نصب باید به جزئیات همه این ابزارها و فریمورک‌ها مسلط باشد. مثلاً باید بدانند چطور نسخه‌های مختلف .NET. یا Java روی یک ماشین اجرا می‌شود با توجه به این پیچیدگی ریسک خطا و اشتباه در طول نصب و راهاندازی بالا می‌رود.

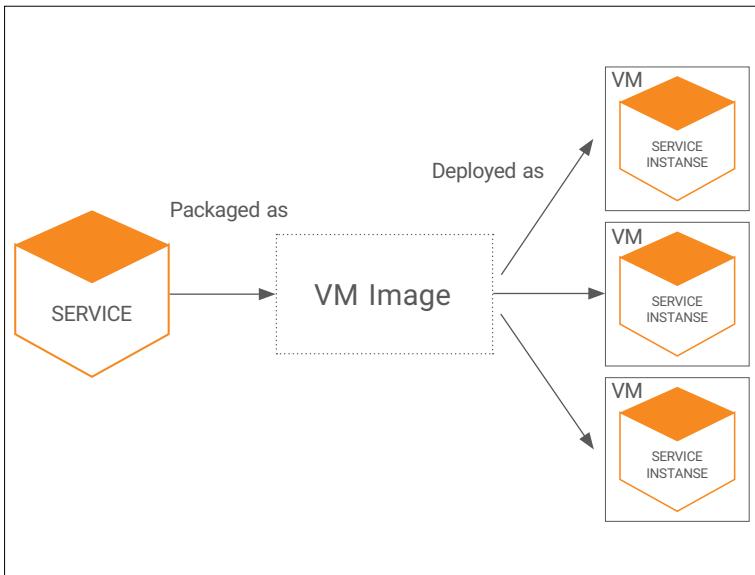
همانطور که مشاهده می‌کنید با اینکه این روش بسیار ساده و آشنا است، مشکلاتی در پی دارد که باعث می‌شود نیاز پیدا کنیم از روش‌های دیگری برای انجام این کار استفاده کنیم.

■ **روش دوم: نصب یک سرویس به ازای هر میزبان:**

هنگامی که از این روش برای نصب و راهاندازی سرویس‌ها استفاده می‌کنیم، یک نسخه از سرویس را روی یک میزبان کاملاً مجزاً نصب می‌کنیم. برای پیاده‌سازی این الگو دو روش مختلف قابل تصور است. روش اول استفاده از ماشین مجازی برای هر نسخه از سرویس است و دومین روش استفاده از کانتینرها است.

■ **نصب هر سرویس روی یک ماشین مجازی:**

هنگامی که از این روش استفاده می‌کنیم، ابتدا از نرم‌افزار خود یک خروجی تهیه می‌کنیم و روی یک ماشین مجازی نصب می‌کنیم. سپس از ماشین مجازی نسخه پشتیبان تهیه کرده و هر زمانی که نیاز به نصب یک نسخه جدید از سرویس داشته‌باشیم، نسخه‌ای جدید از ماشین مجازی را اجرا می‌کنیم. پیاده‌سازی این روش را در تصویر صفحه بعد مشاهده می‌کنید.



هنگام انتخاب این سناریو از ابزارهای متفاوتی مثل- Teamcity, Octopus, Jenkins, Packer و ... می‌توانید استفاده کنید تا روال انتشار نرم‌افزار و نصب آن روی ماشین مجازی و تهیه Image از ماشین مجازی را به صورت اتوماتیک پیاده‌سازی کنید که بررسی این روش‌ها خارج از حوصله این مبحث است و در صورتی که تمایل داشته باشید می‌توانید مطالب تخصصی این حوزه را مطالعه نمایید.

بزرگترین مزیت استفاده از این روش ایزووله بودن کامل نسخه‌های اجرایی هر سرویس است. در این روش هر نسخه از سرویس مقدار ثابتی CPU و RAM در اختیار دارد که نمی‌تواند از حد مجاز خود تخطی کند. مزیت دیگری که برای استفاده از این روش می‌توان نام برد، تبدیل شدن ماشین مجازی به یک جعبه سیاه کامل است. بعد از اینکه این جعبه سیاه

آماده شد دیگر نیازی نیست در مورد اجزا داخلی آن و چگونگی تنظیم شدن آن‌ها اطلاعاتی داشته باشیم. پس تیم زیرساخت به سادگی می‌تواند یک نسخه از ماشین مجازی را بدون هیچ دردرسی اجرا کند. فقط کافی است برای انجام این کار منابع کافی در اختیار تیم زیرساخت باشد.

اما در کنار این مزایا استفاده بد از منابع زیرساختی بزرگترین ضعف استفاده از این روش است. شما به ازای هر نسخه از نرم‌افزار خود باید هزینه‌هایی برای زیرساخت نرم‌افزاری مثل سیستم‌عامل و آنتی ویروس و ... در نظر بگیرید.

مشکل دیگری که برای این روش قابل تأمل است، کند بودن نصب نسخه‌ی جدیدی از نرم‌افزار است. هر چند نصب و راهاندازی یک نمونه از Image بسیار سریع است اما هنگامی که نسخه جدیدی از نرم‌افزار را بخواهیم اجرا کنیم مراحل نصب و راهاندازی کند و زمانگیر است.

هنگامی که سرویس ما دچار اختلال شود یا به هر دلیلی نیاز به اجرای مجدد داشته باشد، روال راهاندازی مجدد با توجه به اینکه از سطح سیستم عامل انجام می‌شود زمانگیر و کند است که این ایجاد هم به نوبه خود حائز اهمیت می‌شود.

با توجه به اینکه در این شرایط معمولاً تیم توسعه روال آماده‌سازی و ایجاد نسخه پشتیبان از ماشین مجازی را به عهده دارد، کارهای خارج از وظیفه و بعض سخت و زمانگیر به تیم توسعه تحمیل می‌شود.

■ نصب و راهاندازی به کمک کانتینرها:

بعد از بررسی دو روش قبل نوبت به کانتینرها می‌رسد. به جرات می‌توان

گفت بهترین راه انتشاره میکروسرویس‌ها استفاده از کانتینرها است که البته نکاتی که گفته می‌شود همه با این فرض است که از داکر استفاده کنیم. در این روش یک داکر فایل ایجاد می‌کنیم که مراحل آماده‌سازی زیرساخت و اجرای سرویس در این داکر فایل به صورت مستند ثبت می‌شود. حال هر زمانی که نیاز باشد به سرعت و با اجرای یک دستور نرم‌افزار ما تبدیل به یک Image داکر می‌شود و آماده استفاده می‌شود. هر زمانی که نیاز داشته باشیم می‌توانیم به سرعت یک نسخه از Image را اجرا می‌کنیم.

استفاده از این روش هم مزایا و معایب خاص خودش را دارد. اما مهم‌ترین مسئله این است که مزایای این روش بر معایب آن می‌چربد. در حوزه مزیت اگر بخواهیم بررسی کنیم، تمامی مزایایی که برای ماشین‌های مجازی بیان کردیم جزء مزیت‌های این روش توزیع و نصب محسوب می‌شود. در کنار این مزایا عیب‌هایی که برای استفاده از ماشین‌های مجازی بیان کردیم هم دیگر وجود ندارد. یعنی در این روش دیگر مجبور نیستیم به ازای هر سرویس هزینه نصب و نگهداری سیستم‌عامل و آنتی‌ویروس را هم به سیستم تحمیل کنیم. در استفاده از این روش تیم توسعه به سادگی مراحل انجام کار نصب و راه‌اندازی را در داکر فایل مستند می‌کند و هر فردی که فقط با داکر آشنا باشد به سادگی می‌تواند بدون نیاز به هیچ دانش خاصی از نرم‌افزار ما سیستم را آماده اجرا کند.

اما با همه مزایایی که در توزیع و نصب با استفاده از کانتینرها وجود دارد معایبی نیز می‌توان برای این روش متصور شد. با اینکه سال‌ها

از شروع به کار داکر می‌گذرد و این بستر بلوغ خوبی پیدا کرده است، اما توسعه‌دهندگان و مهندسین زیرساخت هنوز به خوبی با این ابزار آشنا نیستند و تعداد افرادی که به خوبی می‌توانند از این ابزار استفاده کنند کم است.

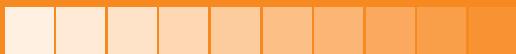
جمع بندی:

نصب و راهاندازی میکروسرویس‌ها چالش بزرگی در استفاده از این معماری است که در این مطلب سعی کردیم با برخی از این چالش‌ها و راهکارهای حل مشکلات صحبت کنیم. در این قسمت بررسی کردیم نصب و راهاندازی چندین سرویس روی یک سرور چه مزایا و معایبی دارد و راه حل جایگزین آن یعنی استفاده از VM‌ها و کانتینرها را نیز مختصررا مورد بررسی قرار دادیم. اساساً این مرحله از کار معمولاً به عهده توسعه دهنده‌های نرم‌افزار نیست و معمولاً دوستان زیرساخت این وظیفه را به عهده دارند، اما با توجه به اینکه باید در مورد نحوه انجام کار تصمیم‌گیری کنیم، سعی شد در این مطلب نکاتی در این رابطه بیان شود.

هر گاه خبرهای بد را به عنوان یک نیاز به تغییر و نه یک خبر منفی پذیرفتید، شما از آن شکست نخوردهاید، بلکه چیزهای تازه‌ای از آن آموخته‌اید.

بیل گیتس

فصل نهم: مهاجرت به میکروسرویس



■ کار را متوقف کنید

■ جداسازی Back-end و Front-end

■ استخراج سرویس

مقدمه:

مهاجرت از Monolith به Microservice یکی از روش‌های مدرن‌سازی فرایند کاری است که هم در دنیای نرم‌افزار و هم در زندگی روزمره انجام می‌شود. حتماً همه ما تجربیاتی روی مهاجرت از فریمورکی قدیمی به جدید داشته‌ایم، یا در زندگی روزمره برای خود ما یا یکی از بستگان فرایند به روز رسانی محیط زندگی اتفاق افتاده، که این به روزرسانی می‌تواند بازسازی منزل، خرید اسباب جدید یا خرید خانه جدید باشد. به هر حال با توجه به اینکه این یک فرایند تکراری است، احتمالاً تجربیات مشترکی نیز در این زمینه وجود دارد که می‌توانیم از این تجربه‌ها استفاده کنیم.

قبل از شروع کل مطلب باید به این نکته توجه کنیم که انواع روش‌های مهاجرت وجود دارد که یکی از آن‌ها Big Bang است. در این روش شما کل نرم‌افزار قدیمی را کنار می‌گذارید و از ابتدا شروع به بازسازی نرم‌افزار با فریمورک و معماری جدید می‌کنید. به طور جد تاکید می‌کنم که از این کار دوری کنید. این روش انجام مهاجرت به شدت ریسک بالایی دارد و احتمال شکست پرخواسته در این شرایط بسیار بالا می‌رود. استاد مسلم حوزه نرم‌افزار مارتین فاولر در این زمینه می‌گوید: «تنها چیزی که در فرایند Big Bang تضمین شده‌است، Big Bang است.» هیچ تضمین دیگری وجود ندارد.

به جای این کار بهتر است این فرایند را یک مهاجرت تدریجی در نظر بگیریم و رفته بخش‌هایی از نرمافزار خود را به معماری جدید مهاجرت دهیم. در این شرایط تا مدتی بخش‌هایی از نرمافزار میکروسرویس است و بخش‌هایی کماکان به صورت Monolith کار می‌کند و کم کم به بخش‌های جدید افزوده می‌شود و بخش‌های قدیمی از بین می‌روند تا زمانی که کل اپلیکیشن به Microservice مهاجرت می‌کند و بخش‌های قدیمی ناپدید می‌شوند. اگر مثالی از دنیای واقعی بخواهیم بیاوریم مانند وقتی که خانه را بازسازی می‌کنیم بهتر است در خانه باشیم و مدتی از این اتفاق به آن اتفاق برویم تا اینکه یک شبه خانه را خراب کنیم برای بازسازی و تا مدت‌ها برای زندگی جایی نداشته باشیم.

این روش مهاجرت را در اصطلاح اپلیکیشن‌های خفه‌کننده می‌گویند. این اصطلاح از درختان خفه‌کننده جنگل‌های بارانی گرفته شده است. این گیاهان به پایه‌های درختان می‌چسبند و خود را از تنه درختان بالا می‌کشند تا به روشنایی خورشید برسند و آنقدر رشد می‌کنند و جلوی نور خورشید را برای درختان می‌گیرند تا درختان می‌میرند و جنگلی از گیاهان خفه‌کننده ایجاد می‌شود. ما اینقدر میکروسرویس‌هایمان را روی پایه‌های Monolith می‌سازیم تا در نهایت Monolith نابود شود.

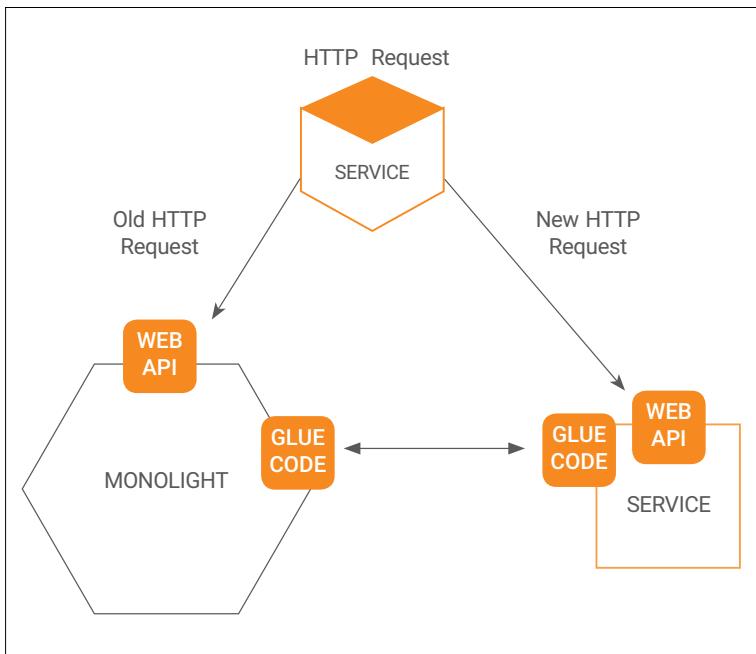


رشد بر پایه‌های درخت میزان

حال بباید انواع روش‌های پیاده‌سازی این روش را بررسی کنیم.

۲. روش اول - کار را متوقف کنید:

قانون اول حفره‌ها به ما می‌گویند زمانی که در یک چاه‌گیر افتادید اول از همه کندن را متوقف کنید. اگر نرم‌افزاری دارید که مدیریت آن برای شما تبدیل به کابوس شده است حتماً این نصیحت را بپذیرید و ادامه فرایند را متوقف کنید. در این شرایط هر زمانی که شما ویژگی جدیدی پیاده می‌کنید نباید آن را به پروژه Monolith خود اضافه کنید. شما باید همه کارهای جدید خود را با معماری جدید ایجاد کنید. در تصویر صفحه بعد نحوه انجام این کار را مشاهده می‌کنید.



همانطور که در تصویر مشاهده می‌کنید در کنار بخش Monolith و Microservice دو عنصر دیگر نیز نیاز داریم تا خدمات را به صورت کامل ارائه دهیم. ابتدا به مازولی به عنوان Request Router نیاز داریم که عملکردی شبیه به API Gateway دارد که در فصل‌های قبل مشاهده کرده‌ایم. در این مازول تصمیم‌گیری می‌شود درخواست ورودی به کدام یک از بخش‌های برنامه باید ارسال شود. فصل بعدی که اگر با توسعه کامپوننت محور آشنای باشید قطعاً با آن آشنا هستید، GLUE CODE است. تا زمانی که دو اپلیکیشن با هم وجود دارند ممکن است که به خدمات یک دیگر نیاز داشته باشند و انجام این خدمات به کمک Glue Code می‌شود که می‌تواند هم در

و هم در Monolith ما وجود داشته باشد. در این شرایط اگر Microservice مثلا نیاز به داده‌ای که در اختیار Monolith است داشته باشد، این کار را از طریق CODE GLUE‌ها انجام می‌دهد. هنگامی که این کار را از داده‌ها داشته باشد، به سه روش مختلف crosservice ما نیاز به این داده‌ها داشته باشد، می‌تواند این کار را انجام دهد.

استفاده از API‌های فراهم شده توسط Monolith

دسترسی مستقیم به دیتابیس Monolith

نگهداری نسخه داده‌های خاص و به روز رسانی آن‌ها به کمک دیتابیس Monolith

با توجه به اینکه Domain Model ما معمولاً Domain Model اختصاصی خود را دارد، و Glue Code از آن‌لوده شدن این مدل با Domain Model موجود در برنامه Monolith جلوگیری می‌کند آن Anti-Corruption Layer موجود در Glue Code کار ترجمه و تبدیل مدل‌ها به یکدیگر نیز گفته می‌شود. عملاین از آن‌لوده شدن این مدل با Domain Model موجود در برنامه Monolith جلوگیری می‌کند آن Anti-Corruption Layer را انجام می‌دهد. اولین بار در کتاب آقای Eric Evans DDD Surrounded معرفی شد و بعداً در مقاله‌ای با عنوان by Legacy Software تشریح شد و پیشنهاد می‌کنم اگر علاقه‌مند به این موضوع هستید این مقاله جذاب را از دست ندهید.

شاید توسعه Anti-Corruption Layer کار سختی به نظر برسد و عدم تعهد به توسعه آن باعث پیچیدگی‌های در انجام کار شود، اما باور دنید اگر می‌خواهید بدون دردسر بتوانید از این چالش خارج شوید به آن نیاز دارید. توسعه بخش‌های جدید به صورت سرویس‌های جدید و سبک مزایایی به همراه دارد که عبارتنداز:

■ جلوگیری از رشد Monolith و افزایش مشکلات
■ امکان توسعه و انتشار و توزیع کاملا مستقل سرویس‌های جدید
■ تجربه بهبود تدریجی شرایط با معرفی میکروسرویس‌ها
اگر کمی در این روش دقیق شویم، متوجه می‌شویم که این روش هیچ‌کدام از معایب Monolith‌ها را به طور مستقیم رفع نمی‌کند و برای رفع آن‌ها باید کار را با سایر روش‌ها ادامه دهیم.

۳- روش دوم - جداسازی Back-end و Front-end

یکی از روش‌های خورد کردن یک Monolith جداسازی لایه Front-end است. اغلب نرم‌افزارهای Enterprise از سه بخش عمده به شرح ذیل تشکیل شده‌اند:

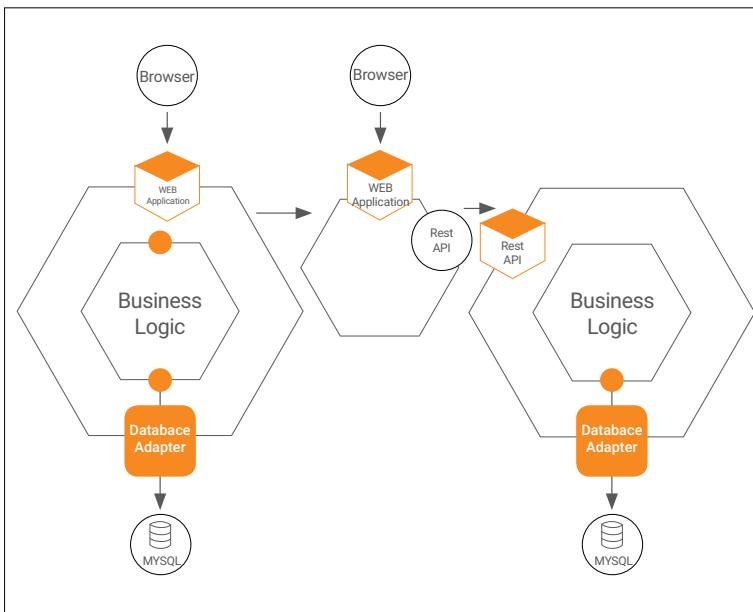
■ **لایه ارائه:** هر نرم‌افزاری نیاز به ارائه خدمات و دریافت داده دارد که این کار اغلب به صورت صفحات HTML یا API انجام می‌شود.

■ **لایه منطق:** قلب نرم‌افزار که مسئولیت محاسبه و منطق نرم‌افزار را به عهده دارد این لایه است.

■ **لایه داده‌ها:** برای تعامل با زیرساخت و ذخیره و بازیابی داده‌ها از این لایه استفاده می‌کنیم.

اگر معماری داخلی نرم‌افزار خوب داشته باشیم، معمولاً تفکیک خوبی بین لایه ارائه و لایه‌های منطق و داده‌ها وجود دارد. اگر این کار را به خوبی انجام داده باشیم، لایه منطق خدمات خود را از طریق تعدادی سرویس در اختیار لایه ارائه قرار می‌دهد و این کار باعث می‌شود بتوانیم یک Monolith را به دو قسمت Back-end و Front-end تقسیم کنیم. در

یک نرم افزار عملیات مربوط به لایه ارائه انجام می شود و در نرم افزار دیگر لایه های منطق و داده قرار می گیرند. بعد از این جداسازی لایه می تواند از طریق توابع Remote به عملیات موجود در لایه منطق دسترسی داشته باشد که نحوه انجام این کار را در تصویر پایین مشاهده می کنید.



- تقسیم یک نرم افزار به این روش مزایای زیر را به دنبال دارد.
 - قابلیت توسعه و توزیع هر کدام از آنها به صورت مستقل.
 - ایجاد یک API خوب برای سایرین تا در صورت نیاز بتوانند برای نرم افزار لایه ارائه ایجاد کنند.
- هرچند با جدایی این دو قسمت از هم مزایای زیادی به دست می آوریم،

اما هنوز مشکلات فراوانی باقی مانده است که برای حل آن‌ها باید به راه خود ادامه دهیم.

۴_ روش سوم - استخراج سرویس:

سومین کاری که می‌توانیم انجام دهیم این است که ماژول‌های اصلی سیستم را تشخیص دهیم و هر کدام را به یک سرویس مجزا منتقل کنیم. با جدایی هر ماژول و خارج کردن هر ماژول بخشی از Monolith خورد می‌شود. بعد از انجام این کار یا به طور کل مشکل Monolith رفع می‌شود یا صورت مسئله‌ما به اندازه کافی شکسته شده و ساده‌تر می‌توانیم آن را حل و فصل کنیم.

۴_ اولیت بندی ماژول‌ها برای تبدیل به سرویس:

یک نرم‌افزار Monolith که سال‌ها در حال توسعه بوده‌است ممکن است شامل ده‌ها یا صدها ماژول متفاوت باشد که همگی کاندیدای تبدیل شدن به سرویس مجزا هستند. تعیین اولین ماژولی که قرار است تبدیل به سرویس شود همیشه یک چالش بزرگ است. یکی از راهکارهای اولویت‌بندی ماژول‌ها انتخاب ماژول‌های ساده‌تر برای مهاجرت است. با این روش هم طعم میکروسرویس را می‌چشیم و هم با فرایند مهاجرت از Monolith به Microservice آشنا می‌شویم.

بعد از اینکه مدتی بر مبنای سادگی ماژول‌ها را جدا کردیم می‌توانیم روش خود را عوض کرده و به سراغ ماژول‌هایی برویم که ارزشمندی بیشتری برای ما ایجاد می‌کنند و سادگی یا پیچیدگی کار را در اولویت بعدی قرار دهیم.

در مرحله بعد شاید بهتر باشد باز هم استراتژی خود را تغییر دهیم و برای ادامه کار مازولهای را انتخاب کنیم که معمولاً تغییرات بیشتری به سمت آنها می‌آید. با این کار با توجه به اینکه فرایند توسعه و انتشار این سرویس‌ها مستقل از کل اپلیکیشن انجام می‌شود، تیم توسعه شرایط کاری بهتری را تجربه خواهد کرد.

در ادامه اگر خواستیم استراتژی انتخاب خود را تغییر دهیم می‌توانیم به منابع توجه کنیم. مازولهایی که نیازمندی بسیار متفاوتی از سایرین دارند بهتر است زودتر از سایر بخش‌ها جدا شوند. برای مثلاً شاید خوب باشد مازولی که مصرف حافظه زیادی دارد زودتر جدا کنیم تا بتوانیم آن را روی ماشینی که RAM زیادی دارد به طور مستقل اجرا کنیم. با انجام این روش جداسازی، شما اپلیکیشن خود را از نظر مقیاس پذیری بهینه می‌کنید. دقت کنید که این ترتیب اولویت بندی صرفاً پیشنهاد است و شاید در حوزه کاری که با آن سر کار دارید با توجه به شرایط نیاز باشد اولویت بندی دیگری را در نظر بگیرید.

۴_۲_ چگونگی انتقال مازول‌ها:

اولین کاری که در این فرایند باید انجام دهیم تعیین واسطه جهت تبادل داده‌ها بین Monolith و میکروسرویس است. این کار را به این دلیل انجام می‌دهیم که مازولهای مختلف نیاز به تبادل داده‌ها دارند این دو نرم‌افزار Monolith و میکروسرویس به دلایل مختلف ناچار به ارسال و دریافت داده به هم هستند. شاید یکی از سخت‌ترین بخش‌های کار همین تعیین واسطه است، به این خاطر که با توجه به عدم وجود مرزبندی دقیق بعضاً

دسترسی‌های غیرمجازی بین مازول‌ها ایجاد شده است. در برخی موارد انتخاب روش پیاده‌سازی لایه منطق نیز ممکن است پیچیدگی این کار را دو چندان کند. شاید مجبور شویم برای از بین بردن وابستگی‌های اشتباه بین مازول‌ها مقداری کدنویسی کنیم.

بعد از تعیین این واسط و جداسازی کامل مازول‌ها وقت آن می‌رسد که مازول خود را به یک سرویس کاملاً مجزا انتقال دهیم و کمی کدنویسی انجام دهیم تا ارتباط بین Monolith و میکرسرویس ما برقرار شود نرم‌افزار مانند گذشته به کار خود ادامه دهد.

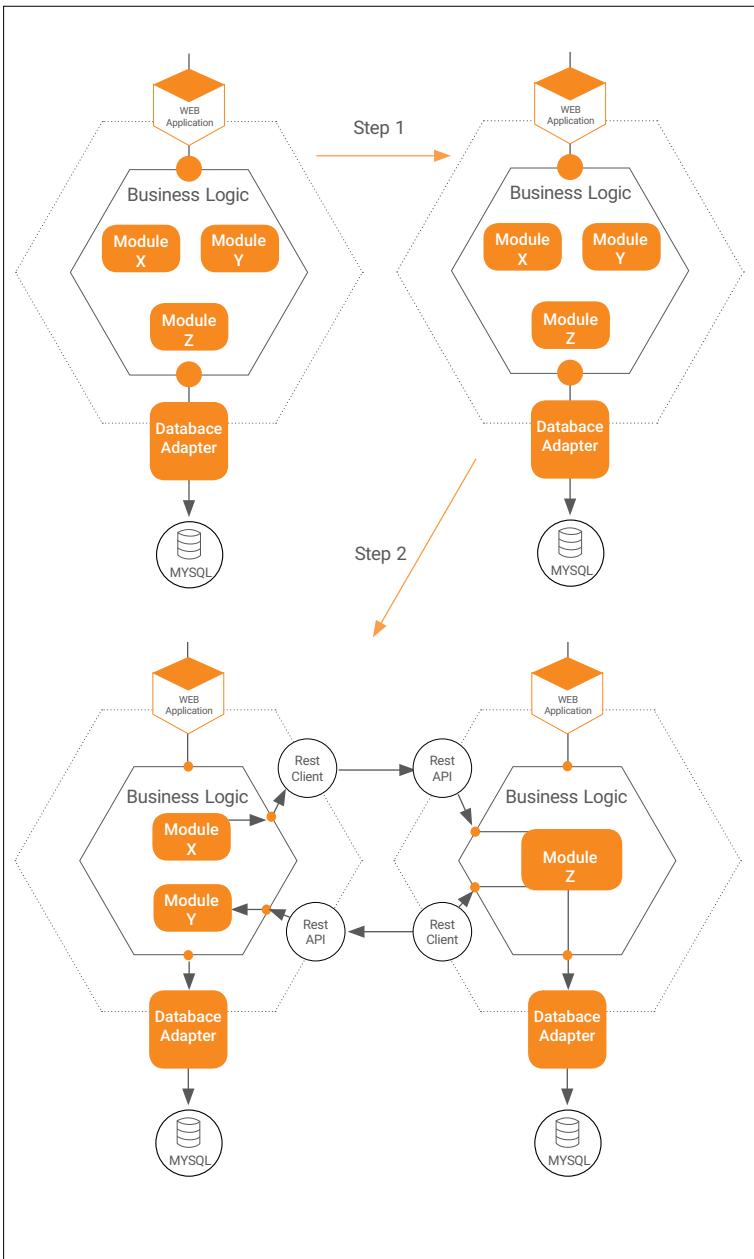
مراحل انجام این کار در تصویر صفحه بعد قابل مشاهده است.



بسیاری از دولوپرهای خوب کدنویسی می‌کنند اما نه به این خاطر که کسب درآمد کنن یا توسط دیگران تحسین بشن، بلکه به این خاطر که کدنویسی باحاله.



لینوس توروالدز



تصویر صفحه قبل مازول Z برای انتقال انتخاب می‌شود. همانطور که مشاهده می‌کنید این مازول خدماتی را از Z دریافت می‌کند و خدماتی نیز به X می‌دهد و بین این سه مازول وابستگی وجود دارد. اولین قدم ما تعریف واسطی برای ارتباط بین Y، X و Z است.

بعد از این کار مازول Z را به سرویسی مجزا تبدیل می‌کنیم و واسطه‌ایی که برای ارتباط بین مازول‌ها تعریف شده را به کمک روال‌های ارتباطی مختلفی که وجود دارد پیاده‌سازی می‌کنیم.

بعد از جداسازی این مازول، حال شما یک سرویس کاملاً مستقل دارید که می‌توانید آن را به تنها‌ی توسعه دهید و نگهداری کنید. ممکن است تصمیم بگیرید با توجه به شرایط رابط آن سرویس را باز نویسی کنید یا اینکه اقدام به Refactor کردن آن سرویس بکنید. اگر تصمیم به باز نویسی سرویس بگیرید واسطی که قبل از طراحی کرده‌اید می‌تواند برای شما نقش Anti Corruption Layer را داشته باشد و از تاثیرگذاری روال توسعه قبلی به سرویس جدید جلوگیری کند. با جداسازی هر مازول شما قدمی به سمت میکروسرویس شدن-Mono-lith خود بر می‌دارید و در نهایت آن غول بی‌شاخ و دم را نابود خواهید کرد.

۵. خلاصه:

فرایند تبدیل یک Monolith به میکروسرویس یکی از کارهایی است که باید برای به روزرسانی سیستم‌ها در صورت نیاز انجام دهیم و برای این کار نباید در ابتدای راه، اقدام به دور ریختن کد موجود کنیم. هر چند ممکن است بعد از گذشت مدتی و جداسازی سرویس‌ها برخی از آن‌ها

را کاملا دور بریزیم و از ابتدا باز نویسی کنیم، اما انجام این کار به طور کلی و در ابتدای راه ریسک بالایی را ایجاد خواهد کرد.

فصل دهم: آشنایی با میکرو فرانت‌اند

(Micro Front-end اول قسمت)



■ سیستم‌ها و تیم‌ها

■ خروجی کار - Front-end

■ ادغام - Integration

■ موضوعات مشترک

مقدمه:

در طول بیش از ۱۷ سالی که در صنعت فناوری اطلاعات و به طور عمده تولید نرمافزار فعالیت کردم شانس این را داشتم تا بعضی الگوهای تکراری را چندین بار در کارهای مختلف مشاهده کنم. کار با تعداد کمی توسعه دهنده در یک پروژه ساده بسیار جذاب و دوست داشتنی است. همه اعضای تیم فهم کلی از همه پروژه دارند.

ویژگی‌های جدید خیلی سریع و ساده به نرمافزار اضافه می‌شوند و صحبت کردن در مورد شرایط و مشکلات پیش رو بسیار ساده و بدون دردرس اتفاق می‌افتد. در صورتی که پروژه موفق باشد به مرور شرایط تغییر خواهد کرد. محدوده پروژه بزرگتر شده و تعداد اعضای تیم بیشتر و بیشتر می‌شود. با این رشد شرایط دیگری هم بسیار آهسته در حال تغییر است. دیگر فهم بخش‌های مختلف برنامه برای توسعه اعضای تیم ممکن نیست.

دانش کسب و کاربخش بخش می‌شود و هر بخش از این دانش نزد یکی از توسعه دهندها و اعضای تیم است. پیچیدگی‌ها بیشتر می‌شود و تغییر در بخشی از نرمافزار موجب تاثیر ناخواسته روی سایر قسمت‌ها می‌شود. گپ و گفتهای دوستانه درباره بخش‌های مختلف نرمافزار تبدیل به جنگ و جدل‌های خسته کننده می‌شود. به جای اینکه تصمیمات خود را با دوستان خود هنگام یک گپ و گفت دوستانه و در حال میل کردن چای و ساقه‌طلایی بگیرید، باید

جلسات رسمی تشکیل دهید و همه اعضای تیم با حالتی کامل رسمی دور هم جمع شوند. معمولاً در نهایت به شرایطی می‌رسیم که دیگر نه کارها پیش می‌رود و نه با افرودن نیروی انسانی جدید به تیم تغییری در روند اجرا رخ می‌دهد.

برای بهبود شرایط معمولاً در این شرایط نرم‌افزار به چندین قسمت شکسته می‌شود. یکی از راهکارهای معمول برای انجام این کار شکستن افقی پروژه و ساختار تیم بر اساس تکنولوژی است. معمولاً تیم‌های دیتابیس، Front-end و Back-end تشکیل می‌شوند و هر تیم مسئولیت بخشی از نرم‌افزار را بر عهده گرفته و کار را پیش می‌برد. به جای استفاده از این روش، Micro Front-end استفاده از شکستن عمودی پروژه را پیشنهاد می‌دهد. هر کدام از این بخش‌ها از پایین‌ترین لایه تا خروجی کاربر توسط یک تیم جداگانه توسعه داده می‌شود. تفاوت این روش با میکروسرویس در لایه آلا است. در این روش توسعه، به جزء سرویس لایه آلا نیز برای هریک از بخش‌های نرم‌افزار توسط تیم تولید می‌شود و نیاز به تولید یک آلا مرکزی را از بین می‌برد. این روش توسعه مزایای زیادی دارد که برخی از آن‌ها عبارتند از:

■ **سرعت بالا در تولید ویژگی‌ها:** هر تیم مسئولیت ۱۰۰٪ یک کار را

به عهده داشته و نیازی به هماهنگی با سایر تیم‌ها کم می‌شود.

■ **به روز رسانی ساده‌تر خروجی:** هر تیم مسئولیت کلیه لایه‌ها را

بر عهده دارد و هر تیم به سادگی و بدون وابستگی به سایر تیم‌ها می‌تواند ابزارها و روش‌های خود را به روز رسانی کند. این ویژگی مانند میکروسرویس‌ها است با این تفاوت که این بار حتی تکنولوژی و روش پیاده‌سازی الا نیز توسط خود تیم انتخاب می‌شود.

■ **تمرکز بیشتر روی مشتری:** هر تیم دقیقاً یک کار را انجام می‌دهد و مستقیماً با مشتری‌های خود در ارتباط است و لایه واسطی بین تیم پیاده‌سازی و مشتری وجود ندارد.

در این قسمت قصد داریم بررسی کنیم چه مشکلاتی توسط Micro Front-end حل خواهد شد. در تصویر صفحه بعد تمامی بخش‌هایی که در پیاده‌سازی Micro Front-end حیاتی است را مشاهده می‌کنید. همانطور که مشاهده می‌کنید یک ساختار و معماری جایگزین در این روش پیشنهاد می‌شود. به همین دلیل است که بخش‌ها و شرایط مختلفی در تصویر زیر مشاهده می‌شود، مثل ساختار تیم‌ها، روش‌های مجتمع‌سازی و ...

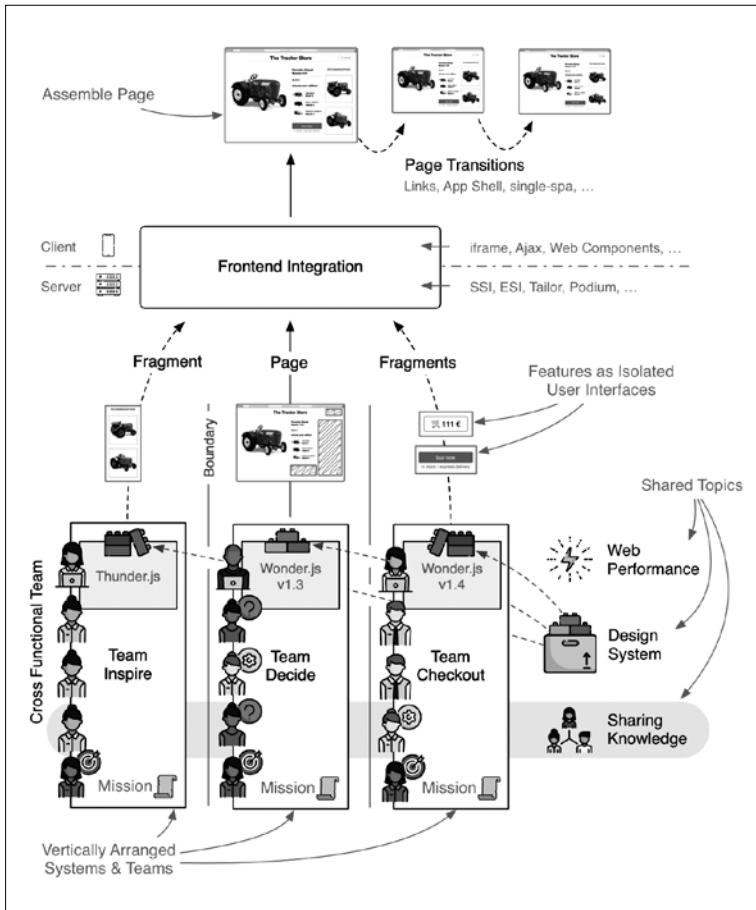
نگران درک کامل این تصویر نباشید، طی چندین فصل این تصویر را از پایین‌ترین لایه تا بالا بررسی خواهیم کرد.



садگی یک برنامه یکی از شرایط قابل اطمینان بودن آن است.



ادسخر دیکسترا



یک دید کلی از Micro Front-end

۲_ سیستم‌ها و تیم‌ها:

همان طور که مشاهده می‌کنید در قسمت پایین صفحه قبل، سه تیم مختلف نرم‌افزاری کنار هم جای گرفته‌اند. هسته اصلی این معماری همین تیم‌ها هستند. هر کدام از سیستم‌ها به صورت کاملاً خودمنحثه

عمل می‌کنند و این به این معناست که حتی اگر یکی از سیستم‌های همکار هم از کار افتاده باشد، باز هم سایر سیستم‌ها به درستی به کار خود ادامه می‌دهند. برای دستیابی به این مهم نیازمند رعایت و داشتن شرایطی هستیم. اول اینکه هر کدام از سیستم‌ها باید پایگاه داده اختصاصی خود را داشته باشد. دوم اینکه هیچ کدام از سیستم‌ها نباید برای انجام وظایف خود به صورت سنکرون تعاملی با سایر سیستم‌ها داشته باشند. هر کدام از این سیستم‌ها به طور کامل در اختیار یک تیم توسعه نرم‌افزار است که این تیم به صورت کامل از پایگاه داده تا ال این سیستم را توسعه و نگهداری می‌کند.

نحوه تعامل این سیستم‌ها در لایه Back-end مسئله‌ای است که در حوزه میکروسرویس‌ها قرار می‌گیرد. در این مجموعه اما به بررسی چالش‌های لایه ال خواهیم پرداخت.

۱_۲_ ماموریت‌های تیم‌ها:

هریک از تیم‌های توسعه نرم‌افزار تخصصی‌هایی دارد که به کمک آن‌ها ارزش‌هایی را برای مشتریان فراهم می‌کنند. برای مثال در یک سیستم تجارت الکترونیک سه تیم با مشخصات و ماموریت‌های زیر را می‌توانیم تصور کنیم:

■ **تیم اکتشاف:** کمک به مشتریان برای یافتن محصولات و خدمات مورد نیازشان.

■ **تیم تصمیم‌گیر:** به مشتریان کمک می‌کند بعد از اینکه محصولات و

خدمات مورد نیاز خود را یافتند از بین موارد مشابه، یکی را که بیش از سایرین برای آن‌ها مناسب است انتخاب کنند.

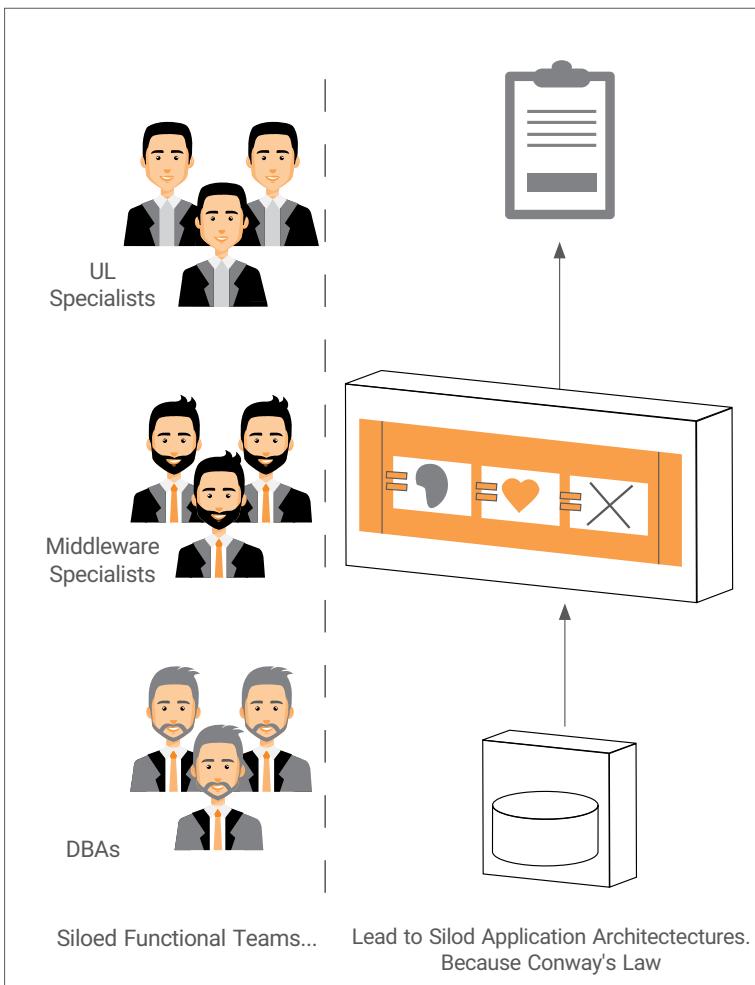
■ **تیم فروش:** کمک به مشتریان در فرایند، خرید، پرداخت و دریافت خدمات و سفارشات.

همانطور که مشاهده می‌کنید، این تیم‌ها از ابتدای کار مشتریان در فرایندهای کسب و کار در مراحل مختلف در کنار آن‌ها هستند تا مشتریان و کاربران به هدف‌شان برسند.

مهمترین مسئله در این قسمت ارائه و تعیین یک هدف مشخص برای تیم‌ها است تا از آن مثل ستاره قطبی برای حرکت در مسیر صحیح استفاده کنند. این هدف مشخص به تیم‌ها کمک می‌کند تا در طول چرخه حیات نرم‌افزار همیشه بدانند چه کاری باید انجام دهند و فعالیت‌های خود را حول چه نیازهایی تعیین کنند.

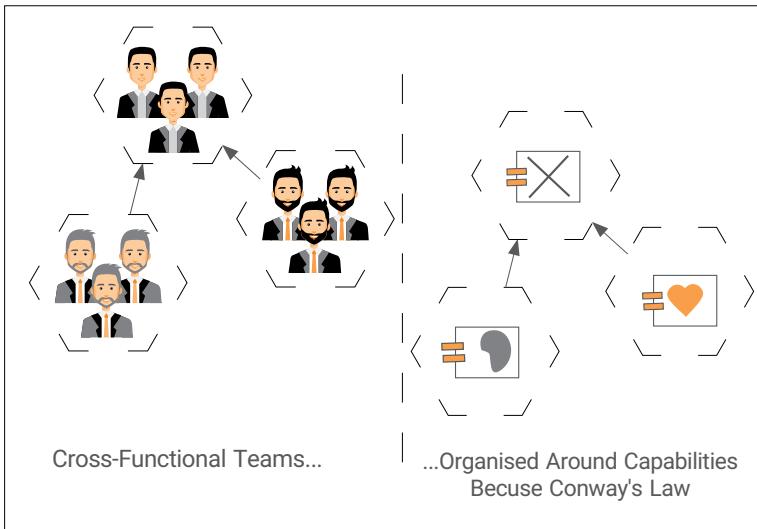
۲_۲- تجمع تیم حول محصول یا تخصص:

در یک مجموعه نرم‌افزاری تیم‌ها به دو شکل مختلف قابلیت چیدمان دارند. حالت اول اینکه تیم‌ها حول توانایی‌هایی خود جمع شوند، مثل تیم‌های UI و Database و Back-end



تشکیل تیم‌ها حول توانایی‌ها و تخصص‌ها

حالت دوم تجمع تیم‌هایی تشکیل شده از اعضایی با تخصص‌های متفاوت برای به انجام رساندن کامل یک ماموریت.



تشکیل تیم‌هایی از تخصص‌های مختلف برای یک هدف خاص

در نگاه اول به نظر می‌رسد ساختار تیم‌ها به شکل اول کاملاً صحیح است. افراد مختلف با توانایی‌های یکسان زبان مشترکی دارند و می‌توانند در مورد مسائل روز حوزه کاری خود با هم تعامل کنند. ضمن اینکه در صورتی که با مشکلی مواجه شوند، مثلاً همه توسعه دهنده‌های Front-end با هم مشورت کرده و به نتیجه مطلوب می‌رسند. در این شرایط به نظر می‌رسد اگر هر تیمی کار خود را به درستی و کمال انجام دهد حاصل جمع بندی کارهای همه تیم‌ها یک محصول کامل و بی‌عیب و نقص باشد.

در دنیای واقعی اما ساختار تیم‌هایی به شکل بالا عملکرد درستی نخواهند داشت. تیم‌هایی که از عملکرد تیم قبلی خود به درستی آگاهی ندارند و از آینده و مورد استفاده کارکرد خود نیز به خوبی آگاه نیستند معمولاً

خروجی‌های ناقص و بلا استفاده‌ای دارند. به همین دلیل است که در سال‌های اخیر، ساختار تیم‌ها تغییر یافته و به جای تجمع حول تخصص، افراد با تخصص‌های متفاوت حول محصولات و ماموریت‌های مختلف گرد هم می‌آیند و معمولاً حاصل کار با توجه به دید کاملی که همه افراد تیم از عملکرد خود دارند، با کیفیت‌تر و خلاقانه‌تر است. شاید طراحی تیم‌ها به این شکل موجب این شود که محصول در یکی از قسمت‌ها در بالاترین سطح کیفی خود قرار نداشته باشد، برای مثال از آخرین نسخه از Angular برای تولید خروجی استفاده نشده باشد، اما با توجه به اینکه همه هدف تیم ارائه خروجی با کیفیت و مورد نیاز مشتری است، در نهایت مشتری خوشحال‌تری خواهیم داشت. مهمترین دست آورد چیدن تیم‌ها به این روش این است که با توجه به اینکه همه اعضا با هر توانایی خروجی کار را دریافت می‌کنند، به مرور توانایی این را پیدا خواهند کرد تا خود را جهت رسیدن به هدف نهایی مدیریت و برنامه‌ریزی کنند.

۳_ خروجی کار - :Front-end

این قسمت نتیجه همه کارهای تیم است یعنی جایی که کار به خروجی می‌رسد و هر تیم به طور کاملاً مجزا مسئول به ثمر رساندن کار خود است. این به این معناست که هر تیم مسئول تولید مقادیری کدهای HTML و CSS و JavaScript برای ارائه خروجی کار خود است. احتمالاً تیم‌ها برای اینکه لذت بیشتری از زندگی ببرند و عذاب دنیا و آخرت کمتری را تجربه کنند برای انجام این کارها سراغ فریمورک‌هایی می‌روند. اما با توجه به ماهیت جداگانه این تیم‌ها هیچ اجبار و تضمینی برای

انتخاب ابزار و فریمورک مشترک بین تیم‌ها نیست. هر تیم با توجه به توانایی و علاقه و نیازهای خود فریمورک خود را انتخاب می‌کند.

۱_۳_ مالک صفحه:

بیایید کمی در مورد صفحات نرم‌افزار صحبت کنیم. به سراغ مثال قبلی خود یعنی یک سیستم تجارت الکترونیک می‌رویم. نرم‌افزار ما صفحات زیر را خواهد داشت:

■ **صفحه اصلی:** نمایش لیستی از کلیه محصولات، تبلیغات، تخفیفات

... ۹

■ **صفحه لیست و جستجوی کالا:** نمایش لیست کالاهای.

■ **صفحه جزئیات کالا:** نمایش جزئیات و اطلاعات کامل و دقیق محصولات و خدمات به مشتریان.

■ **سبد خرید:** نمایش کالاهای انتخاب شده.

■ **پرداخت:** نمایش فاکتور و دریافت تاییدیه برای پرداخت آنلاین و ارسال سفارش.

■ **تایید پرداخت:** نمایش فاکتور نهایی و پیام تشکر بابت خرید.

با توجه به این تعریف‌ها و تیم‌هایی که قبلاً تعریف کردیم یعنی اکتشاف، تصمیم‌یار و فروش احتمالاً می‌توانید حدس بزنید که هر کدام از این صفحات متعلق به کدام تیم است! صفحه اصلی و لیست متعلق به تیم اکتشاف، صفحه جزئیات کالا متعلق به تیم تصمیم‌یار و صفحات سبد خرید، پرداخت و تایید پرداخت متعلق به تیم فروش می‌باشد. خوب با شرایطی که تا اینجا بررسی کرده‌ایم هر کدام از این صفحات

توسط یک تیم توسعه و منتشر می‌شود و در نهایت در آدرس‌هایی در وب، هاست می‌شوند و کاربر می‌تواند به کمک لینک‌هایی که دارد بین این صفحات حرکت کند. (البته در دنیای واقعی پیچیدگی‌هایی وجود دارد که باعث می‌شود همیشه کار به این سادگی نباشد و نیاز به دانش و مهارت زیادی برای پیاده‌سازی خواهیم داشت).

۳_۲ قطعات - Fragments

مفاهیم موجود در صفحات همیشه کامل و جدا نیستند. در اصل بخش‌هایی در صفحات داریم که کاملاً مشترک است مثل هدر و فوتر صفحات و نمی‌خواهیم این بخش‌ها توسط تیم‌های متفاوت چندین بار توسعه داده شوند.

در کنار مسئله بالا توجه به این نکته نیز لازم و ضروری است که همیشه یک صفحه کاملاً مربوط به یک هدف خاص نیست. برای مثال در صفحه‌ی جزئیات که تیم تصمیم یار مسئول آن است ممکن است بخشی وجود داشته باشد که لیستی از کالاهای مرتبط برای تبلیغ و ترغیب به خرید وجود داشته باشد که مسلمًا پیاده‌سازی این قسمت به عهده تیم اکتشاف است. یا سبد خرید جمع و جوری را در همه صفحات یک فروشگاه مشاهده می‌کنید که خلاصه‌ای از وضعیت سبد خرید را نمایش می‌دهد که مسئول این قسمت نیز تیم فروش است.

همانطور که مشاهده می‌کنید یک صفحه به طور کامل متعلق به یک تیم نیست و اینجاست که با مفهوم Fragment آشنا می‌شویم. تیم‌هایی که مسئول یک صفحه هستند برای تکمیل مسئولیت و

ماموریت خود تصمیم می‌گیرند از خروجی‌های سایر تیم‌ها استفاده کنند. بعضاً برای استفاده از خروجی یک تیم نیاز است اطلاعاتی از وضعیت صفحه جاری در اختیار تیم هدف قرار بگیرد مثلًا برای نمایش کالاهای مرتبط توسط تیم اکتشاف در صفحه جزئیات کالا نیاز است که تیم تصمیم یار شناسه کالای جاری را در اختیار تیم اکتشاف قرار دهد. در برخی شرایط نیز هیچ وابستگی وجود ندارد، مثلًا برای نمایش سبد کالا در سایر صفحات تیم فروش نیاز به هیچ داده‌ای خارج از حوزه کاری خود ندارد. نکته حائز اهمیت در اینجا این است که تیم مالک Fragment صفحه در این شرایط اطلاعی از شرایط تیم فراهم آورنده و چگونگی انجام کار ندارد.

۴_ ادغام - :Integration

در نهایت همه این بخش باید با هم ترکیب شوند و ساختار نهایی صفحات و نرم‌افزار را ایجاد کنند.

۴_۱ ادغام _Front-end

فرایند دریافت Fragment‌ها و قراردادن آن‌ها در محل‌های صحیح داخل صفحه در این قسمت انجام می‌شود. در اصل این کار توسط تیمی که مسئول صفحه است انجام نمی‌شود بلکه آن‌ها در فرایند تولید صفحات خود بخش‌هایی را برای سایرین در نظر می‌گیرند و این بخش‌ها را در صفحات خود علامت گذاری می‌کنند یا به اصطلاح در صفحات خود Placeholder‌هایی را قرار می‌دهند که Fragment‌ها باید در این قسمت‌ها قرار بگیرند.

در نهایت سرویس یا بخشی مسئولیت دریافت صفحات و یافتن Frag-ment‌های مناسب برای هر Placeholder و سرهم کردن آن‌ها را بر عهده دارد. به طور کلی این فرایند را می‌توان به دو دسته تقسیم کرد که با توجه به نیاز خود می‌توانید یکی یا ترکیبی از هر دو را انتخاب کنید:

■ **انجام به صورت Server Side:** تجمعیع با SSI, ESI, Tailor و Podiom

■ **انجام به صورت Client Side:** استفاده از Web iFrame, Ajax

Component

در کنار روش مجتمع‌سازی که انتخاب می‌کنید باید به دنبال روشی برای ارتباط برقرار کردن نیز باشید. برای مثال هنگامی که کاربر در صفحه جزئیات محصول رنگ یا اندازه محصول را تغییر می‌دهد بخش مربوط به پیشنهاد نیز باید از این تغییر باخبر شود تا در صورت نیاز کالاهای پیشنهادی را تغییر دهد.

۴_۲_انتقال بین صفحات:

تا اینجای کار به بالاترین سطح برنامه یعنی Front-end رسیده‌ایم. به هر حال نیاز داریم تا از صفحه‌ای به صفحه دیگر پیمایش کنیم، این کار را می‌توانیم به کمک لینک‌های عادی و بارگذاری کامل صفحه انجام دهیم یا به کمک روش‌های هوشمندانه‌ای به صورت کاملا Client-Side و بخش به بخش صفحه‌ها را به روزرسانی کنیم.

۵_موضوعات مشترک:

هرچند که Micro Front-end تلاش می‌کند تیم‌های غیر وابسته و خودمختار کوچک با مسئولیت‌ها و عملکردهای مشخص ایجاد کند، اما

به هر حال مطالب مشترکی وجود دارد که باید برای همه تیم‌ها حل و فصل شود.

۱_۵_ بهینگی:

داشتن صفاتی که حاصل ترکیب چندین قسمت و کارکرد مختلف است، موجب می‌شود تا نیاز داشته باشیم به مسئله بهینگی مصرف منابع توجه ویژه‌ای داشته باشیم. برای مثال باید به دنبال راهکاری باشیم که در صورتی که فریمورک مشترکی بین چند تیم استفاده می‌شود از بارگذاری چندین باره آن جلوگیری کنیم.

۲_۵_ طراحی سیستم:

با اینکه بخش‌های مختلف توسط تیم‌های مختلف و کاملاً مستقل توسعه داده می‌شوند اما باید به اینکه نکته توجه داشته باشیم که در نهایت همه کارها در قالب یک کل در اختیار کاربر قرار می‌گیرد و داشتن ساختاری منسجم یکی از ملزمات سیستم است. مثلاً اگر قرار است از لوگوها و شکل‌هایی در نرم‌افزار استفاده شود، باید مجموعه مشترکی بین تیم‌ها وجود داشته باشد.

۳_۵_ اشتراک دانش:

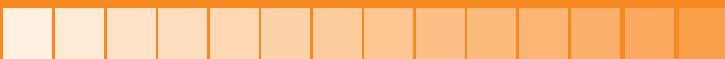
با اینکه باید تلاش کنیم که تیم‌های خود مختار و مستقلی داشته باشیم، اما باید در مقابل عدم انتشار دانش نیز مقاومت کنیم. در کل باید در حالی که جدایی کامل تیم‌ها را در نظر می‌گیریم از انجام چند باره کارهای پایه‌ای و فراهم کردن چندین باره زیرساخت‌های مشترک مثل Logging و ... جلوگیری کنیم. ایجاد زیرساخت‌های مشترک بین تیم‌ها می‌تواند

یکپارچگی و مدیریت محصول را ساده‌تر کرده و در نهایت تمرکز تیم‌ها بر فرایندهای کسب و کاری را بهبود بخشد. ایجاد شرایطی برای اشتراک نظر و انتقال دانش و تشریک مساعی نیز می‌تواند همدلی کلی تیم‌ها و دید اعضای تیم‌ها نسبت به شرایط کسب و کار را بهبود ببخشد.

۶ خلاصه:

در این فصل سعی کردیم به طور کلی با مفهوم Micro Front-end در آشنایی. و مزایا و معایب استفاده از این روش و چالش‌هایی که با آن‌ها روبرو هستیم را بشناسیم. در فصل‌های بعدی جزئیات بیشتری را در مورد این روش توسعه بررسی خواهیم کرد.

فصل یازدهم: مشکلاتی که حل می‌کنیم: (Micro Front-end دوم)



- سرعت بالا در افزودن ویژگی‌های جدید
- حذف گلوگاه Front-end
- توانایی تغییر
- توانایی تغییر توانایی تغییر
- معایب Micro Front-end
- افزونگی

مقدمه:

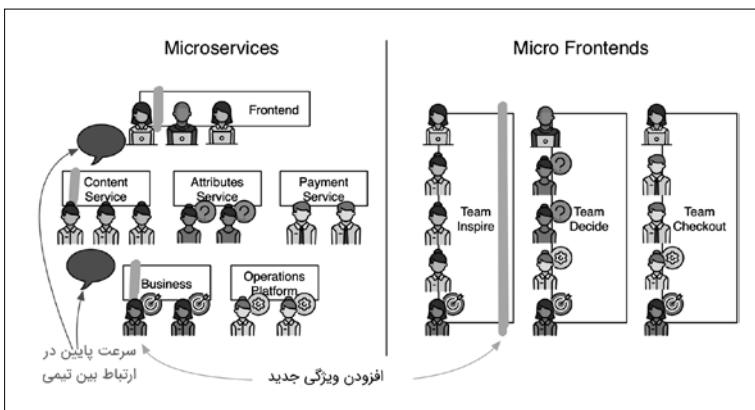
در فصل قبل از این مجموعه به طور اجمالی با آشنایی دیدیم نحوه مدیریت تیم و تقسیم وظایف به چه شکلی در این روش انجام می‌شود. در نهایت بررسی کردیم ترکیب صفحات در این روش به چه شکلی انجام می‌شود. حال که ایده بهتری در مورد چیستی Micro Front-end ها داریم، بباید با هم نگاهی دقیق‌تر به مزایای این روش توسعه و دستاوردهای آن داشته باشیم.

۲_ سرعت بالا در افزودن ویژگی‌های جدید:

به جرات اولین دلیل تیم‌های فنی از انتخاب Micro Front-end برای توسعه نرم‌افزارها، سرعت بالای توسعه در این روش است. در یک تیم معمول که از میکروسرویس‌ها برای توسعه نرم‌افزار استفاده می‌کند تیم‌های زیادی برای افزودن یک ویژگی جدید درگیر می‌شوند. فرض کنید تیم کسب و کار تصمیم می‌گیرد برای افزایش قدرت خود در بازار و تبلیغات بهتر نوع جدیدی از بنرها را ایجاد کند. برای این کار آن‌ها به سراغ تیم سرویس محتوا می‌روند و ایده‌های خود را با آن‌ها مطرح می‌کنند. و از آن‌ها می‌خواهند ساختار داده‌ها را به گونه‌ای تغییر دهند تا پیاده‌سازی ایده بنر جدید امکان پذیر باشد. اما مسلماً تغییر ساختار داده‌ها به تنها‌ی کاربرد ندارد و باید جلساتی بین تیم محتوا و Front-end برگزار شود تا در مورد نحوه انتقال داده‌ها به Front-end و نمایش و مدیریت آن تصمیم‌گیری کنند. جلساتی تشکیل می‌شود و ویژگی‌ها

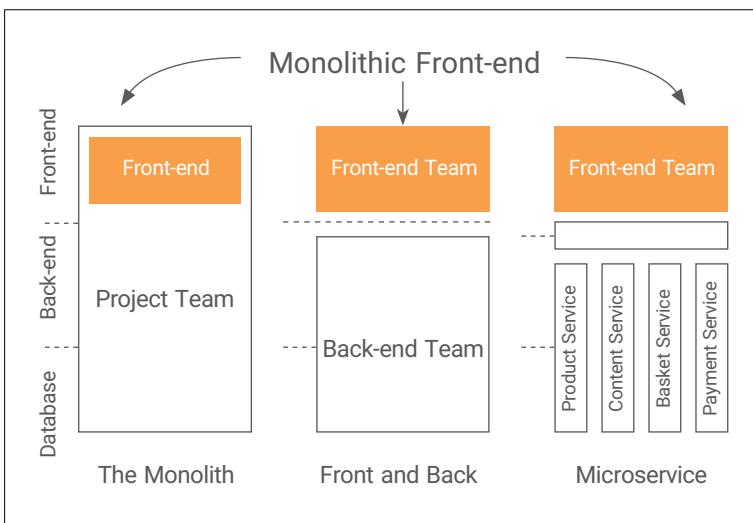
لیست می‌شود. هر تیم مسئولیت‌های خود را در نظر می‌گیرد و در طول اسپرینت‌های آتی سراغ پیاده‌سازی‌های خود می‌رود. اگر همه چیز به خوبی پیش برود بعد از چند اسپرینت محصول آماده بهره‌برداری است. اما اگر کار دچار مشکل شود نیاز به چندین و چند جلسه بین تیمی است تا تیم‌ها عملکردها و نیازهای خود را با سایرین تنظیم کنند.

اینجا به سادگی یکی از تفاوت‌های روش Micro Front-end قابل مشاهده است. با اینکه کاری که قرار است انجام شود، تغییری نمی‌کند، اما حضور افراد درگیر در انجام یک کار در یک تیم باعث می‌شود، نیازها خیلی سریع‌تر تشخیص داده شود و به جای برگزاری جلسات بین تیمی زمانگیر، با چند گپ و گفت درون تیمی کار به خوبی پیش برود. درون تیم‌ها روابط بسیار صمیمی‌تر است و از جو رسمی جلسات بیرون تیمی خارج می‌شویم. نیاز به کارهای جانبی رسمی مثل صورت جلسات و ... کاسته می‌شود و در نهایت نیازی نیست تا به اعضای تیم دیگری در مورد اولویت کارها و ... توضیح دهیم و توافق کنیم.



۳_ حذف گلوگاه :Front-end

در اغلب روش‌های توسعه نرم‌افزار مطرح برای Scale کردن Front-end فکری نشده است. در تصویر بعدی مشاهده می‌کنید که در سه روش مرسوم در توسعه نرم‌افزار، حتی وقتی از میکروسرویس‌ها استفاده می‌کنیم کماکان یک لایه Front کاملاً Monolithic داریم.



در End-to-End Front-end حتی Micro Front-end هم به چندین قسمت کوچک تقسیم می‌شود که مزایای زیر را برای ما به همراه خواهد داشت:

- تست ساده‌تر
- محدوده عملکرد کوچک‌تر و دقیق‌تر
- سورس کد کوچک‌تر
- توسعه و نصب کاملاً مجزا

- فهم و نگهداری ساده‌تر
- بازنویسی و بهبود ساده‌تر
- انتخاب تکنولوژی تخصصی و مناسب
- احتمال شکست کمتر و محدود کردن حیطه شکست پروژه
- بیایید کمی با جزئیات بیشتری این مزایا را بررسی کنیم.

۴_ توانایی تغییر:

به عنوان یک توسعه‌دهنده و مهندس نرم‌افزار طی این سال‌ها این مورد را به خوبی دریافت‌هایم که یادگیری مستمر و به کارگیری دانش جدید تنها راه پیشرفت در این حرفه است. این نیاز به تغییر و یادگیری و به روزرسانی هرچه به کاربر نهایی نزدیک‌تر باشد، بیشتر احساس می‌شود و مسلماً نزدیک‌ترین بخش به کاربر نهایی لایه Front-end است. ابزارها و تکنولوژی‌ها در این لایه به سرعت تغییر می‌کنند. سال ۲۰۰۵ با معرفی AJAX جهان متحول شد و فکر می‌کردیم به حد نهایی پیشرفت رسیده‌ایم، اما فاصله بین ریلیزهای فریمورک‌ها و تکنولوژی‌های Front-end این نظریه را تایید نمی‌کند. برای مثال نیم نگاهی به فاصله زمانی انتشار نسخه‌های Angular داشته باشید تا احساس دقیق‌تری به سرعت انتشار داشته باشید. از سال ۲۰۰۵ که تنها زیبایی و مرتب بودن یک صفحه کفايت می‌کرد تا امروز که طراحی و پیاده‌سازی Front-end تبدیل به کاری بسیار حساس و تخصصی شده‌است روزگار زیادی سپری نشده‌است. از روزگاری که Front-end صرفاً بخشی بی‌اهمیت از رزومه توسعه دهنده‌ها بود تا امروز که پیاده‌سازی Front-end نیاز به یک رزومه کامل و قوی دارد تنها

۱۵ سال گذشته است که این سرعت پیشرفت واقعاً اعجاب آور است.
(البته با سرعت رشد دلار قابل مقایسه نیست و خدا رو شکر به لطف
دوستان ما سرعت‌های بالاتری تجربه کردیم).

این روزها دیگر Front-end فقط HTML و CSS نیست و یک توسعه
دهنده متخصص Front-end باید مطالب زیادی بداند مثلاً باید بداند
طراحی Responsive چیست و چگونه انجام می‌شود. باید حداقل یک
فریم ورک مثل Angular را مسلط باشد و با React و VU نیز باید آشنا
باشد. با استانداردهای روز و تغییرات دائمی آن‌ها و پشتیبانی آن‌ها
توسط بسترهای مختلف باید کاملاً آشنا باشد.

۴_۱ میراث بد

کدهای مشکل دار و قدیمی همیشه جز معضلات یک تیم نرم‌افزاری
است و این مشکل در Front-end نیز قابل اجتناب نیست. هر چه قابلیت
تغییر و بهینه‌سازی در این لایه را بالاتر ببریم، می‌توانیم به جای مجبور
کردن نیروهای ارزشمند به تغییر و نگهداری کدهای قدیمی از آن‌ها در
کارهای جدید و بهتر بهره ببریم که باعث بالارفتن کیفیت سیستم و
انگیزه اعضای تیم نیز می‌شود. هر چه سیستم بزرگ‌تر و گره خورده‌تر
باشد این به روزرسانی سخت‌تر خواهد شد. برای مثال Github نزدیک
به ۱ سال زمان صرف کرد تا کل سیستم خود را از JQuery خلاص کند.
وقتی در دنیای رقابتی امروز زندگی می‌کنید باید بتوانید خیلی سریع
ویژگی‌هایی متناسب با تکنولوژی روز را به نرم‌افزار خود بیافزایید. این
مهم است که افزودن ویژگی‌های جدید با تکنولوژی روز و جدید انجام

شود. استفاده از تکنولوژی روز به این معنی نیست که شما مجبور هستید برای هر تغییری کل سیستم را باز نویسی کنید. بلکه باید بتوانید روی موج فناوری حرکت کنید.

۴_ تصمیمات و تاثیرات کوچک:

توانایی اینکه در یک تیم کوچک برای بخشی از کار بتوانید تکنولوژی جدید و دلخواه خود را انتخاب کنید بدون اینکه نیاز باشد با جامعه بزرگی طرف شوید و مجبور به اعمال تغییرات بسیار زیاد در بخش‌های زیادی از نرم‌افزار باشید ارزشی است که به سادگی به دست نمی‌آید. این امکان را به شما می‌دهد تا تصمیمات بزرگی را در تیم‌های کوچک و محلی بگیرید.

فرض کنید که بخشی از نرم‌افزار بسیار حساس است و باید کاملاً از باگ عاری باشد. اما ماهیت جاوا اسکریپت امکان خطایابی را مشکل می‌کند. بنابر این تیمی که مسئول انجام این کار است تصمیم می‌گیرد به سراغ TypeScript رفته از قابلیت کامپایل شدن آن و یافتن برخی خطاهای و جلوگیری از بروز برخی خطاهای دیگر استفاده کند. اگر روش عادی توسعه را استفاده کرده باشیم باید کل برنامه از این روش استفاده کند که خوب سربار یادگیری و بازنویسی زیادی را به سیستم تحمیل می‌کند. اما در Micro Front-end تنها تیمی که مسئولیت حیاتی دارد می‌تواند بهای این کار را بپردازد.

۵_ مزایای عدم وابستگی:

خود مختاری یکی از دستاوردهای عدم وابستگی است. به سادگی و

بدون نیاز به تایید سایر تیم‌ها یک تیم می‌تواند عملکرد خود را تصحیح کند یا تغییر دهد. هرچند در صورتی که همه تیم‌ها از یک روش و تکنولوژی هم استفاده کنند باز هم این خود مختاری در موقعي می‌تواند گره‌گشا باشد.

هنگامی که می‌گوییم در روش Micro Front-end یک سیستم بدون وابستگی است، نیاز است که یک بخش همه نیازهای خود را تامین کند. یعنی HTML, CSS, Script, Data Fragment را برای کار کردن صحیح داشته باشد. با این روش تیم‌ها می‌توانند یک را بدون اینکه نیاز به تغییر بخش دیگری باشد تغییر دهند و منتشر کنند و نرم‌افزار به سادگی به روز می‌شود.

شاید با خود فکر کنید که تولید Front-end در کنار Back-end و اینکه هر سرویس به جای خروجی دادن داده‌ها باید یک مازول کامل همراه با آن را برای خروجی ارسال کند کار سنگینی است که باعث خواهد شد سرویس‌ها از وظیفه اصلی خود و سبک بودن دور شوند. اما باید به این نکته دقت کنیم که این روش همراه میکروسرویس‌ها استفاده می‌شود که خود سرویس‌های کوچک و قابل توزیع ایجاد می‌کند و در صورتی که نیاز باشد می‌توان بسیار راحت یک سرویس را Scale کرد و این Scale شدن Front-end را هم در بر می‌گیرد. ممکن است با خود فکر کنید اینکه برای یک صفحه لازم است چندین فریمورک بارگذاری و استفاده شود و بخش‌هایی که می‌تواند به سادگی مشترک باشد چندین و چند بار دانلود می‌شود کمی غیر بهینه است و می‌تواند ما را دچار مشکل کند که فکر اشتباهی نیست. البته برای این مشکلات

راهکارهایی در نظر گرفته شده است ولی بررسی و حل این مشکلات در مباحث این کتاب نمی‌گنجد، البته در آینده در سایت نیکآموز مقالاتی در این زمینه منتشر خواهیم کرد.

۶_ معایب :Micro Front-end

مثل هر روشی استفاده از این روش نیز معایبی دارد و نمی‌توان آن را بدون عیب و مدینه فاضله دانست و باید با دانستن مزایا و معایب این روش در موقع صحیح از آن استفاده کنیم.

۱_۶_ افزونگی:

شاید از هر فرد تازه‌کاری در دنیای فناوری اطلاعات بپرسید بزرگترین اتفاقی که باید از آن اجتناب کنیم چیست به شما پاسخ دهد افزونگی. این افزونگی در بخش‌های مختلف به شکل‌های متفاوت حذف می‌شود. در دیتابیس‌های رابطه با نرم‌افزاری. در برنامه‌نویسی شی‌گرا با ارثبری و ... هر توسعه دهنده‌ای می‌داند بخش‌های تکراری کد را باید در قالب یک ماثول مشترک توسعه دهند و در اختیار تمامی قسمت‌ها بگذارند. با این روش هم کدهای کمتری می‌نویسیم و هم تغییر و نگهداری سریع‌تر و کم خطاطری را تجربه خواهیم کرد. به عنوان یک توسعه دهنده در این سال‌ها مغز و چشم ما یادگرفته تا تکرارها را پیدا کند و سریع برای رفع آن‌ها به دنبال راه حل باشد. وقتی چندین تیم تلاش می‌کنند کارهای کاملاً مستقل ایجاد کنند تکرار، بخش غیرقابل اجتناب سیستم است. باید بدانیم هر دستاوردهی بهایی دارد و بهای استقلال هم بعضاً تکرار است.

۶_ ناهمگونی و عدم تجانس:

قبل از این گفتیم که هر تیم می‌تواند به سادگی تکنولوژی و ابزار دلخواه خود را انتخاب و استفاده کند. خیلی هم خوب و عالی. اما واقعاً نیاز داریم که این‌چنین جهنمی از تکنولوژی را برای خود ایجاد کنیم؟ در این روش اگر توسعه دهنده‌ای قرار باشد از تیمی به تیم دیگر برود تقریباً هیچ چیز با خود نمی‌برد. با توجه به تفاوت حیطه فعالیت قاعده‌تا بیزینس را به خوبی نمی‌شناسد و در این شرایط تنها داشته توسعه‌دهنده دانش فنی است که باز آن هم بی‌فایده می‌شود. همین چند روز پیش در حال بررسی پروژه‌ای مربوط به یکی از سازمان‌های بزرگ بودم که در آن به طور همزمان از EF, Dapper, NHibernate, Angularjs, Angular8, MSMQ, RabbitMQ, NServicebus, MassTransit و ... استفاده شده بود. واقعاً دیدن پروژه چیزی بیش از سرگیجه و به این نتیجه رسیدن که تیم توسعه بیشتر به دنبال تست و یادگیری بوده تا انجام کار نتیجه دیگری نداشت.

۷_ نتیجه:

مثل هر روشی توسعه Micro Front-end هم مزایا و معایب خاص خود را دارد. در این سری مسائل سعی می‌کنیم با روش پیاده‌سازی به این روش آشنا شویم و مثل یک ابزار خوب در کنار سایر ابزارها از آن نگهداری کنیم و هر زمان نیاز بود از آن استفاده می‌کنیم نه همیشه.

با یادگیری درست معماري ميكروسرвис،
۵ سال در مسیر حرفه‌اي خود جلو بیفتید.



مدرس:
علیرضا ارومند



لینک صفحه
دوره آموزشی

کد تخفیف
micro

nikamooz;
تجربه، آموزش، آینده

QR را اسکن یا روی لینک کلیک کنید.

با کد تخفیف **micro**,

دوره آموزشی را با **یک میلیون تومان تخفیف** تهیه کنید.