بسم الله الرحمن الرحیم

میکروسرویس

nikamooz
آموزش برنامه نویسی و اجرای پروژه

عنوان: میکروسرویس

نویسنده: مهندس علیرضا ارومند

برنامه‌ریزی و اجرا: شرکت فن آوری اطلاعات نیک آموز

مدیرعامل: مهندس فرید طاهری

انتشارات: کلید آموزش

طراح جلد و صفحه‌آرا: حبیبه دریایی اصل

چاپ اول: ۱۳۹۹

تیراژ: ۱۰۰۰ نسخه

قیمت: ۴۰,۰۰۰ تومان

شابک:

آدرس: تهران، یوسف آباد، خیابان ابن سینا، خیابـان ۳۳، پلاک ۲۹، طبقـه دوم

تلفن: ۱۷ ۰۰ ۰۷ ۹۱ - ۰۲۱ فکس: ۵۹ ۸۰ ۴۲ ۲۸ - ۰۲۱

سایت: www.nikamooz.com - www.nikamooz.pro

درباره نویسنده:

مهندس علیرضا ارومند دارای فوق لیسانس نرم‌افزار از دانشگاه شیراز است. وی هم اکنون به عنوان Technical Manager شرکت داتین (وابسته به فناپ) در حوزه توسعه سیستم‌های بانکداری مشغول به فعالیت است.

ایشان یکی از افراد پیشرو در حوزه‌های معماری نرم‌افزار، مهندسی نرم‌افزار و NET Core. در ایران است. مهندس علیرضا ارومند به عنوان Technical Manager در شرکت نیک آموز نیز فعالیت می‌کند و با مدیریت این تیم پروژه‌های بسیار بزرگی را با موفقیت در نیک‌آموز انجام داده‌است.

فعالیت‌های تخصصی مهندس علیرضا ارومند:

- طراحی و توسعه فریمورک تخصصی بر پایه ASP.NET Core
- مدرس دوره‌های مهندسی نرم افزار و NET Core.
- مشاور و معمار ارشد نرم‌افزار در حوزه NET Core.
- مدرس دوره‌های تخصصی NoSQL و DDD
- تدریس دوره معماری میکروسرویس

پیشگفتار

دنیای نرم‌افزار به سرعت در حال گسترش است و شرکت‌های بسیاری کسب و کارهای خود را بر اساس نرم‌افزار بنا نهاده‌اند. با توجه با بازار کسب و کار و نیاز شرکت‌ها به رقابت با یکدیگر، لازم است نرم‌افزارها توانایی رقابت را ایجاد کنند. برای داشتن نرم‌افزاری قابل رقابت باید چندین ویژگی مختلف در هر نرم‌افزاری وجود داشته‌باشد. نرم‌افزارها باید قابلیت تغییر سریع داشته‌باشند، پایداری بالایی داشته و امکان پاسخ به تعداد زیادی درخواست را دارا باشند و با توجه به ماهیت نرم‌افزارها که دیگر ابزارهای جانبی نیستند و بنیان کسب و کارها را تشکیل می‌دهند باید امنیت بالایی نیز داشته‌باشند.

در چنین شرایطی همه ارکان توسعه نرم‌افزار تحت تاثیر قرار می‌گیرند. یکی از این ارکان اساسی معماری نرم‌افزار 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 روش‌های توسعه نرم‌افزار توسط بسیاری از شرکت‌های بزرگ نرم‌افزاری مورد استفاده قرار می‌گیرد. اما در این شرایط چالش‌های جدیدی به دنیای توسعه نرم‌افزار وارد می‌شود. چالش‌هایی از قبیل تعیین حوزه هر سرویس، مدیریت داده‌های توزیع شده، برقراری امنیت در معماری جدید و استفاده از ابزارها و روش‌های جدید بخشی از این پیچیدگی‌ها و چالش‌ها است که باید به آن‌ها جواب داده‌شود.

در ادامه در این کتاب سعی شده‌است که این چالش‌ها پاسخ داده

شوند. ابتدا به بررسی دلایل عدم کاربردی بودن معماری‌های سنتی می‌پردازیم و محاسن و راهکارهای میکروسرویس‌ها را برای رفع مشکلات معماری یکپارچه بیان می‌کنیم. در بخش بعد با توجه به ماهیت توزیع شده سیستم‌های میکروسرویس در مورد ارتباط بین سرویس‌های مختلف و چالش‌های ایجاد شده صحبت می‌کنیم. در این بخش در مورد نحوه برقراری ارتباط به صورت Sync و Async گفتگو می‌کنیم و می‌بینیم چطور توزیع شدگی کار با داده‌ها را می‌تواند مشکل و چالش برانگیز کند. برقراری امنیت و انواع روش‌های ایجاد امنیت یکی دیگر از بخش‌های این کتاب است. نصب و راه‌اندازی میکروسرویس‌ها یکی دیگر از چالش‌های دنیای توسعه میکروسرویس است که در این کتاب در مورد آن صحبت به میان می‌آید و در نهایت با هم شرایط مهاجرت از یک معماری یکپارچه به میکروسرویس را بررسی می‌کنیم. آخرین فصل‌های کتاب اما به بررسی مقدماتی Micro Front-endها مربوط می‌شود و در ادامه میکروسرویس‌ها سعی می‌کنیم با دلایل استفاده از این روش توسعه نیز آشنا شویم.

در پایان از همکاران و دوستان خود در مجموعه نیک‌آموز به خصوص جناب آقای مهندس فرید طاهری که در آماده‌سازی و انتشار این اثر همواره در کنار اینجانب بوده‌اند، تقدیر و تشکر می‌کنم.

تقدیم به

همسرم، اسطوره زندگیم، پناه خستگیم و امید بودنم.

پدرم که عالمانه به من آموخت تا چگونه در عرصه زندگی، ایستادگی را تجربه نمایم.

و به مادرم، دریای بی‌کران فداکاری و عشق که وجودم برایش همه رنج بود و وجودش برایم همه مهر.

و به دخترم که همه قشنگی‌های دنیا با لبخدش شروع می‌شود.

فهرست:

فصل اول ـ میکروسرویس چیست؟ ۱۴
دنیا قبل از میکروسرویس به چه صورت بود؟ ۱۴
پیش به سوی جهنم! ۱۶
بهشت گمشده میکروسرویس ۱۹
مزایای میکروسرویس‌ها ۲۲
معایب میکروسرویس‌ها ۲۴

فصل دوم ـ آشنایی با API Gateway ۳۰
تعامل مستقیم Client ها با میکروسرویس ها ۳۳
حل مشکلات با استفاده از API Gateway ۳۵

فصل سوم ـ ارتباط بین سرویس‌ها ۴۴
انواع روش‌های تعامل ۴۵
تعریف APIها ۴۸
تکامل APIها ۴۹
مدیریت بحران هنگامی که بخشی از سیستم دچار مشکل می‌شود......... ۵۱

فصل چهارم ـ تکنولوژی‌های ارتباطی ۵۶
ارسال پیام به صورت Async ۵۷
ارتباط Sync و Request/Response.. ۶۱
سرویس‌های REST ۶۱
مزایای HTTP ۶۲
معایب HTTP ۶۳
ساختار پیام‌ها ۶۳

فصل پنجم ـ آشنایی با Service Discovery ۶۶
آشنایی با Service Registry ۶۸
آشنایی با الگوی Client-Side Discovery ۶۸
آشنایی با الگوی Server-Side Discovery ۷۰
بررسی دقیق Service Registry ۷۲
انواع روش‌های مدیریت سرویس‌ها ۷۴

فصل ششم ـ مدیریت داده‌ها درمیکروسرویس‌ها ۸۰
مشکلات داده‌های توزیع شده در میکروسرویس‌ها ۸۰
توسعه بر مبانی Eventها ۸۵
ویژگی Automicity هنگام استفاده از Eventها ۹۰

فصل هفتم ـ امنیت در میکروسرویس ۹۸
تاریخچه، نحوه امن کردن نرم‌افزارهای یکپارچه ۱۰۰
مبانی کلیدی امنیت ۱۰۴
امنیت مرزهای نرم‌افزار ۱۱۲
امن کردن ارتباط سرویس به سرویس ۱۱۹

فصل هشتم ـ آشنایی با روش‌های انتشار ۱۳۶
نصب چند سرویس مختلف به ازای هر میزبان ۱۳۷
نصب یک سرویس به ازای هر میزبان ۱۳۹
نصب هر سرویس روی یک ماشین مجازی ۱۳۹
نصب و راه‌اندازی به کمک کانتینرها ۱۴۱

فصل نهم ـ مهاجرت به میکروسرویس ۱۴۶
کار را متوقف کنید ۱۴۸

جداسازی Front-end و Back-end ۱۵۱
استخراج سرویس ۱۵۳

فصل دهم _ آشنایی با میکروفرانت‌اند ۱۶۰
سیستم‌ها و تیم‌ها ۱۶۳
خروجی کار Front-end ۱۶۸
ادغام Integration ۱۷۱
موضوعات مشترک ۱۷۲

فصل یازدهم _ مشکلاتی که حل می‌کنیم ۱۷۶
سرعت بالا در افزودن ویژگی‌های جدید ۱۷۶
حذف گلوگاه Front-end ۱۷۸
توانایی تغییر ۱۷۹
مزایایی عدم وابستگی ۱۸۱
معایب Micro Front-end ۱۸۳

فصل اول: میکروسرویس چیست؟

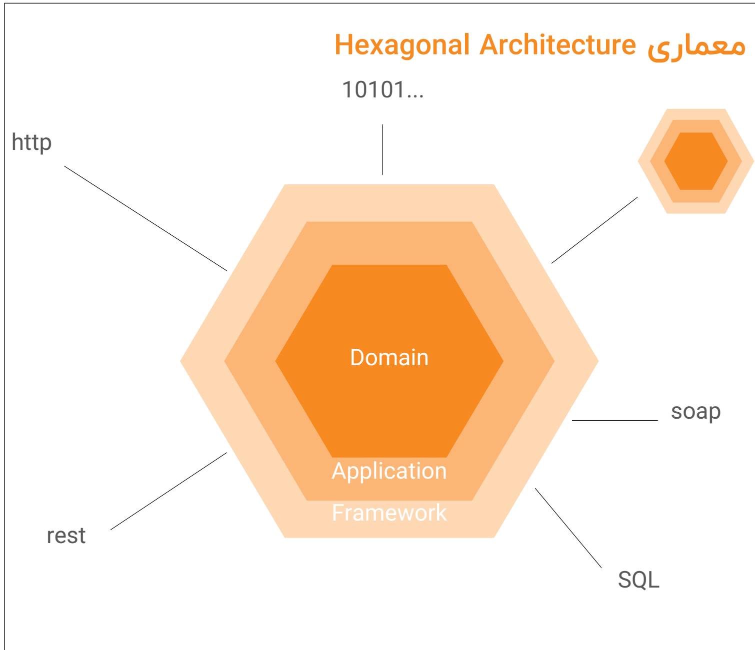
- دنیا قبل از میکروسرویس به چه صورت بود؟
- پیش به سوی جهنم!
- بهشت گمشده میکروسرویس [Microservice]
- مزایای میکروسرویس‌ها
- معایب میکروسرویس‌ها
- جمع‌بندی [میکروسرویس‌ها]

دنیا قبل از میکروسرویس به چه صورت بود؟

فرض کنید شما برای توسعه یک نرم‌افزار دعوت به همکاری شده‌اید. از شما خواسته می‌شود یک CMS خبری توسعه بدهید 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 مشاهده‌ی مطالب سایت را خواهندداشت.

از آنجا که شما یک توسعه‌دهنده حرفه‌ای هستید احتمالا سراغ یک ابزار خوب و یک ساختار خوب و تمیز خواهید رفت، مثلا معماری Hexagonal یا معماری Onion برای توسعه انتخاب می‌کنید و در نهایت یک نرم‌افزار بسیار تمیز توسعه می‌دهید. در همچین شرایطی احتمالا 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 احمقی 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 کاملا ماژولار طراحی و پیاده‌سازی شده‌است اما در واقع کل این ماژول‌ها کاملا وابسته به هم هستند و در قالب یک بسته نرم‌افزاری روی خروجی قرار می‌گیرند. اینکه این یک بسته نرم‌افزاری دقیقا 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 خواهندشد و هر جایی نیاز به نرم‌افزار داشته‌باشید 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 بی‌حکمت نیست“ اگر بخواهیم به زبان برنامه‌نویسی ترجمه کنیم می‌شود ”هیچ سادگی بدون محدودیت نیست.“ هر نرم‌افزار و پروژه‌ای در صورتی که موفق بشود قطعا تمایل به رشد دارد و موفقیت بیشتر پروژه موجب رشد بیشتر پروژه خواهدشد. در نهایت بعد از مدتی پروژه کوچک و ساده ما تبدیل به هیولایی بزرگ و وحشتناک خواهدشد.

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

مشکل اول: با بزرگ و پیچیده‌شدن 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 احتمالا به بن بست

خواهد رسید. بزرگترین مشکل این نرم‌افزارها پیچیدگی بیش از حد این برنامه‌ها است. معمولا 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 اصطلاحا 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 احتمالا این سیستم عظیم بهره‌وری کل سیستم شما را هم پایین خواهد آورد.

مشکل سوم: احتمالا با داشتن یک سیستم بزرگ شما با عدم توانایی در انتشار بهبودها و توسعه‌های کوچک روبرو خواهید شد. مسلما سیستمی که باز کردن آن چند دقیقه طول بکشد، Build شدنش بعضا ۱ ساعت طول خواهد کشید و روال نصب راحتی هم نخواهدداشت. همچنین با توجه به اینکه در زمان نصب احتمالا سیستم از دسترس

خارج خواهدبود و قاعدتاً از دسترس خارج کردن یک سیستم عظیم به خاطر یک تغییر کوچک یا رفع باگ احتمالی جزئی، مقرون به صرفه نیست به همین خاطر نصب نسخه‌های جدید با فاصله‌های زمانی طولانی انجام خواهدشد.

مشکل چهارم: عدم استفاده بهینه از منابع یکی دیگر از مشکلات اساسی توسعه نرم‌افزار به این شکل است. در نرم‌افزارهای متفاوت نیازهای متفاوتی وجوددارد. ممکن است بخشی از برنامه مصرف RAM زیادی داشته‌باشد 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 نخواهدداشت. در نتیجه هنگامی که بخشی با مصرف CPU زیاد ارتقا داده بشود این امکان در اختیار همه بخش‌ها قرار خواهدگرفت و برای همه قسمت‌ها امکان استفاده از این منبع، بی‌رویه و ناصحیح وجود خواهدداشت.

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 خواهدشد. 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 خواهدبود. 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 احتمالا وقتی با یک ابزار، تکنولوژی یا فریمورک جدید مواجه بشوید به جزء حسرت امکانات موجود کار دیگری از دستتان برنمی‌آید. معمولا 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 شده‌است در این ابزار از NET Framework 2.0. استفاده‌شده و برای توسعه وب هم از ASP.NET Web Form استفاده شده‌است و اینقدر سیستم بزرگ و پیچیده شده که در طی این سال‌ها کسی جرات دست زدن به سیستم و تغییر تکنولوژی را نداشته‌است در صورتی که خودتان جای یکی از مدیران پروژه و شرکت بگذارید 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 15-20 نفره به طور میانگین درگیر این پروژه بوده‌است چقدر هزینه تولید همچین ابزاری شده‌است؟ آیا تغییر تکنولوژی با این شرایط امکان پذیر هست؟

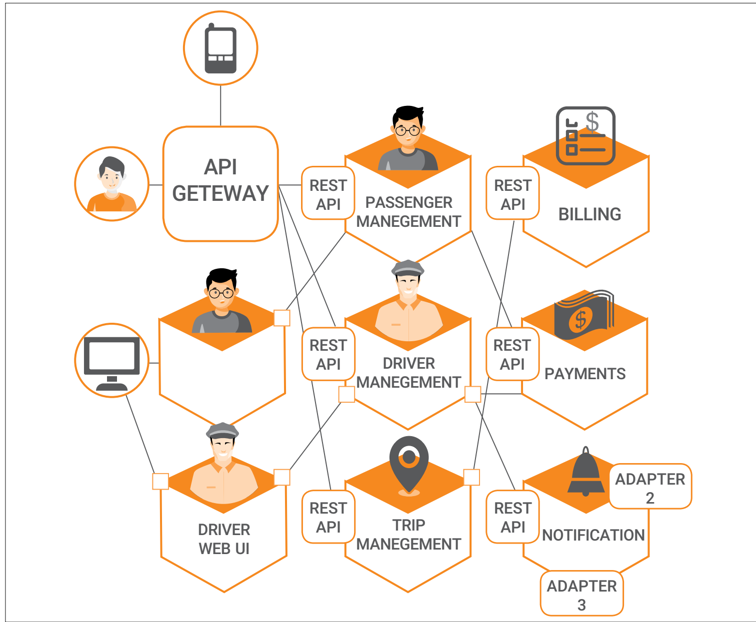
گویا کم کم در حال نزدیک شدن به پایان دنیا هستیم. اما واقعا راه حلی برای این مشکلات نیست؟

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 معماری تمیز مثل Hexagonal یا Onion به طور داخلی استفاده می‌کند. هر مینی‌اپلیکیشن این توانایی را خواهدداشت 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 خواهند 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 ذخیره‌شده در سیستم دسترسی داشته‌باشد. برای ارتباط بین سرویس‌ها راهکارهای مختلفی وجود دارد و اگر با این روش‌ها آشنا نیستید اصلا نگران نباشید. در قسمت‌های بعد در مورد این روش‌ها کامل صحبت خواهیم کرد.

خوب تا اینجا همه چیز به نظر خوب می‌آید اما احتمالا به این موضوع فکر بکنید که اگر تعداد مینی‌اپلیکیشن‌ها زیاد بشود و هر برنامه هم API خودش را ارائه بدهد و برنامه‌ها نیاز داشته‌باشند که با هم ارتباط برقرار کنند تعداد زیادی آدرس بوجود خواهدآمد 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 خواهدشد و در مورد جزئیات این بحث در قسمت آینده کامل صحبت خواهیم‌کرد.

مزایای میکروسرویس‌ها

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

احتمالا 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 خواهدشد. 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. قطعا با کوچک شدن برنامه‌ها فهم و توسعه برنامه‌ها هم به شدت ساده‌تر از قبل خواهدشد.

اغلب شرکت‌های نرم‌افزاری از تعدد Stackهای توسعه نرم‌افزار وحشت دارند و معمولا 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 دقیقا مناسب با نیاز آن‌ها است استفاده کنیم. به راحتی اگر داده‌های ما ساختار گراف داشته‌باشید 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 خواهدشد.

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 Pay for Play قطعا به دست آوردن این همه مزیت بدون هزینه و ایراد نخواهد 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. 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 داشته‌باشد. برای ارتباط بین سرویس‌ها راهکارهای مختلفی وجود دارد و اگر با این روش‌ها آشنا نیستید اصلا نگران نباشید. در قسمت‌های بعد در مورد این روش‌ها کامل صحبت خواهیم کرد.

احتمالا به این موضوع فکر بکنید که اگر تعداد مینی‌اپلیکیشن‌ها زیاد بشود و هر برنامه هم API خودش را ارائه بدهد و برنامه‌ها نیاز داشته‌باشند 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 خواهدشد 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 بگویم بسیار کوچک این سرویس‌ها تاکید شده‌است. 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 ببنید کدام یک از این مواردی که اسم بردیم کوچک به حساب می‌آید. پس تعیین مقیاس برای سرویس‌ها یکی از مشکلات ما خواهد بود. هنگامی که به اندازه سرویس‌ها فکر می‌کنید حتما این نکته را به یاد داشته‌باشید 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 انجام‌دهیم.

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 خواهدداشت و این توزیع‌شدگی 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 داشته‌باشد و نیاز داشته‌باشیم 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 دقت خواهید دید که همین مورد می‌تواند ایرادات زیادی را ایجاد کند. بعضا 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.

جمع بندی [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 دهنده‌ها 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، کاملا 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 همون‌طور که خوندن و نوشتن رو به بچه‌ها یاد می‌دیم، برنامه‌نویسی رو هم یاد خواهیم داد و افسوس می‌خوریم که چرا زودتر این‌کار رو شروع نکردیم.

مارک زاکر برگ

فصل دوم: آشنایی با API Gateway

- تعامل مستقیم 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 نهایتا 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. احتمالا صفحه‌ای خواهید داشت که جزئیات محصولات را نمایش خواهد داد. بیایید نگاهی به صفحه محصولات دیجی کالا 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: فروشندگان، نقد و بررسی، نظرات کاربران، پرسش و پاسخ، محصولات پیشنهادی و ... اگر کمی ریزبین‌باشید احتمالا متوجه بخش‌هایی که من جا 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

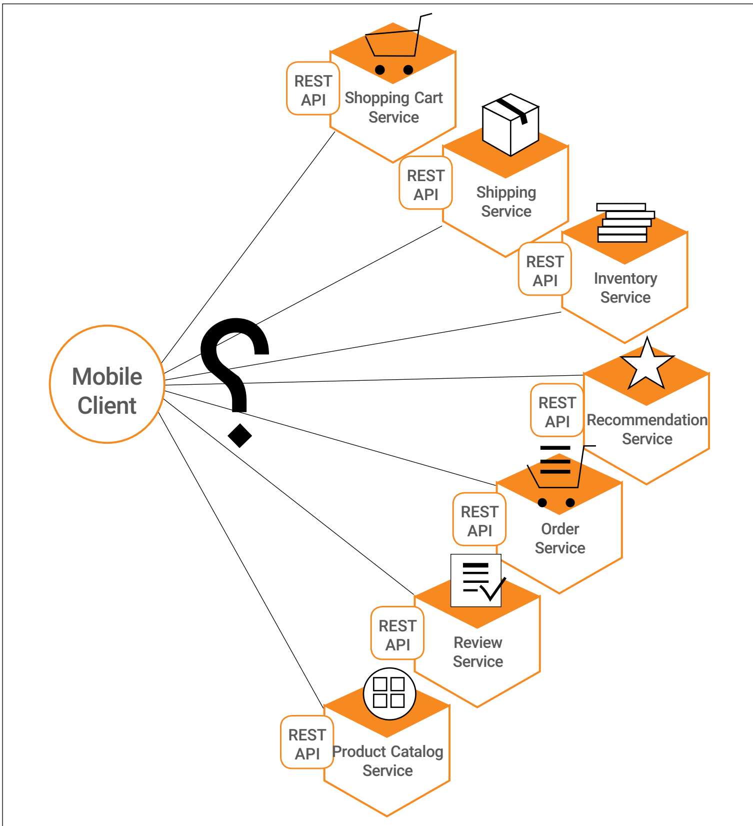
Balancer قرار داشته باشد درخواست توسط Load Balancer مسیریابی شده و به بهترین نسخه اجرایی نرم‌افزار تحویل داده خواهد شد و بعد از عبور از لایه‌های مختلف نرم‌افزار به دیتابیس خواهد رسید و با اجرای کوئری روی چند جدول دیتابیس نتیجه دلخواه آماده شده و در اختیار ما قرار خواهد گرفت.

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 هرکدام
- **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 پشنهادی:** 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 راه‌حل 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 ۱۰ 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 پروتوکل‌های متفاوت برای ارائه API می شود. 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 کاملا محتمل است که در این بین از پروتوکل‌هایی استفاده شود که خیلی Web Friendly نیستند.

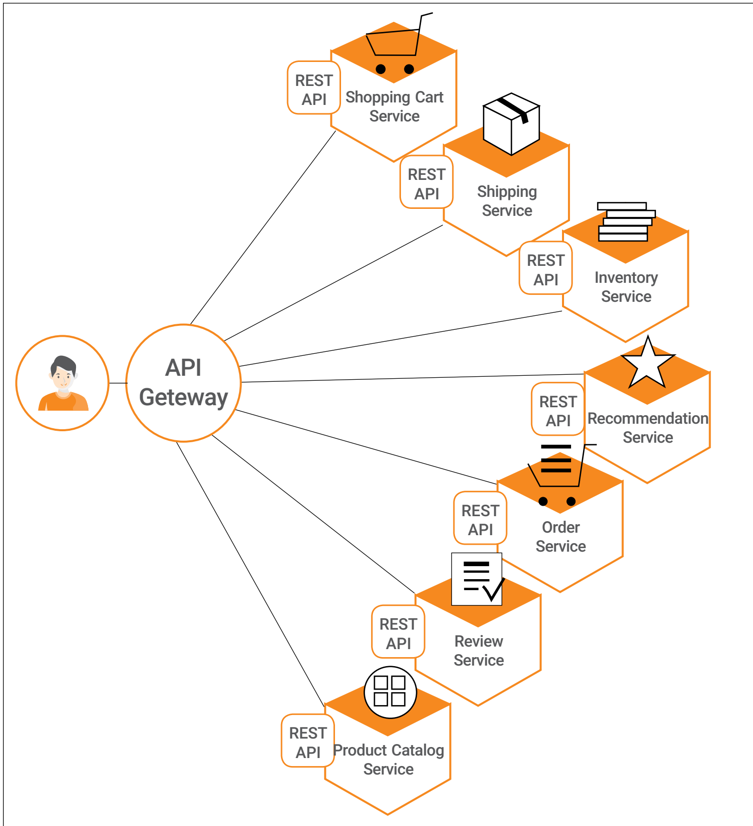
سومین مشکل در این روش مربوط به Refactor کردن میکروسرویس‌ها می‌شود. در طول زمان ممکن است متوجه اشتباهاتی در طراحی خود بشویم و نیاز داشته باشیم که دو یا چند میکروسرویس را با هم ادغام

کنیم یا برعکس یک سرویس را به چندین سرویس تبدیل کنیم. در حالتی که Clientها به طور مستقیم با سرویس‌های ما تعامل کنند احتمالا 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 آب ها 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:

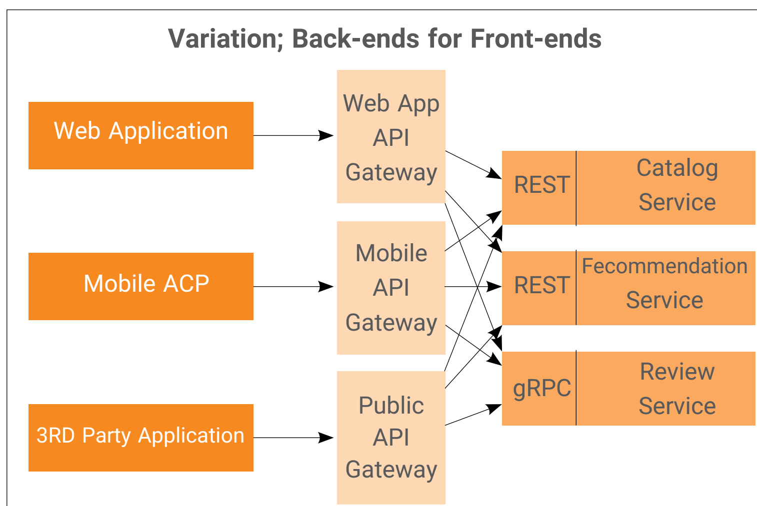
روش بهتری برای تعامل Clientها با میکروسرویس‌ها وجود دارد که آن را با نام API Gateway می‌شناسیم. در این روش تنها نقطه ورود برنامه ما یک Gateway است و راه ارتباطی Clientها با Microserviceها همین Gateway است. در صورتی که با الگوی Facade آشنایی داشته باشید API Gateway عملکردی شبیه به این الگو دارد. در این 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. صدا زدن چندین سرویس مختلف و ترکیب نتنیجه و بازگرداندن نتیجه نهایی مواردی است که باید داخل API Gateway ما کپسوله شود و استفاده کننده نهایی به سادگی و با صدا زدن یک API نتیجه دلخواه خود را دریافت کند.



API Gateway

تمامی درخواست‌های کاربران به API Gateway تحویل داده می شود و در API Gateway مسیریابی به هر سرویس، تعامل با پروتوکل‌های 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 Gatewayهای تخصصی برای هر Client است. مسلما 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 تخصصی داشته‌باشیم که صرفا داده‌های آن Client را فراهم کند.



پیاده سازی Gateway تخصصی

۱ـ۳ـ 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 نداشته‌باشیم (پیدا کردن جایی تا زمانی که آب‌ها از آسیاب بیوفتد). 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 ماژول 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 اختصاصی داشته‌باشیم این احتمال وجود دارد که همین API Gateway 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.

۲_۳_ پیاده‌سازی API Gateway:

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

۱_۲_۳_ مسئله بهره‌وری و مقیاس‌پذیری:

احتمالا 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 مثلا می‌توان به Net Core. و Node.js اشاره کرد.

۲_۲_۳_ 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 احتمالا 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 Reactive Programming راهکار بهینه‌تری نسبت به روش معمول برنامه‌نویسی باشد.

۳_۳_۳_ راهکارهای تعامل:

در یک API Gateway نیاز است با سرویس‌های مختلف تعامل

انجام شود و اصطلاحا Inter-Process Communication انجام شود. سرویس‌های مختلف ممکن است راهکارهای متفاوتی برای تعامل در اختیار ما قرار دهند. ممکن است از روش‌های Async مثل AMQP یا روش‌های 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.

۴_۳_۳_ 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 احتمالا 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.

۵_۳_۳_ مدیریت خطاها:

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. مثلا در سیستم فروشگاه و صفحه جزئیات فروشگاه یکی از سرویس‌ها در دسترس نباشد. مثلا سرویس پیشنهاد کالا در دسترس نباشد. API Gateway باید این قابلیت را داشته‌باشد که در این شرایط به جای اینکه کل

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

۴ـ جمع بندی:

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.

در بسیاری از مواقع افراد نمی‌دانند چه می‌خواهند تا آنکه کسی آنچه را می‌خواهند به ایشان نشان دهد.

استیو جابز

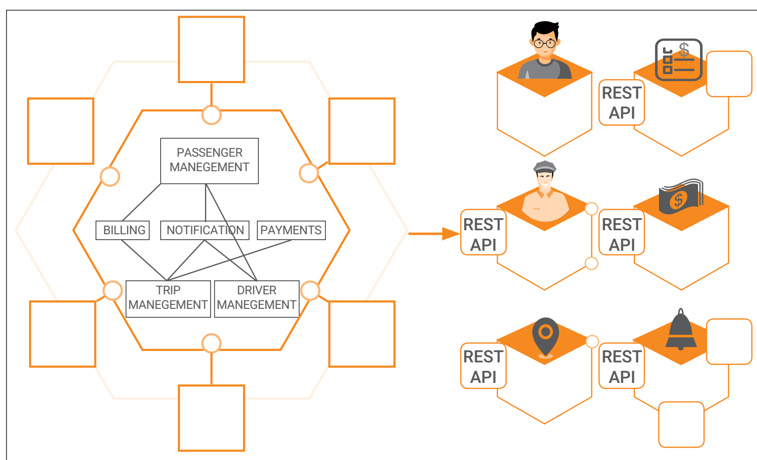
فصل سوم: ارتباط بین سرویس‌ها

- 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 شده‌است 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 کاملا جداگانه و ایزوله اجرا می‌شود. در میکروسرویس‌ها ارتباط بین بخش‌های مختلف اصطلاحا 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 بیایید نگاهی به موارد مختلفی که هنگام طراحی ارتباط بین سرویس‌ها باید مدنظر داشته‌باشیم بیاندازیم.



انواع روش‌های تعامل:

پیش از طراحی و پیاده‌سازی IPC بهتر است در مورد چگونگی برقراری ارتباط بین سرویس‌ها کمی تامل 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 دقیقا 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 بودن ارتباط است.

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

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

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

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

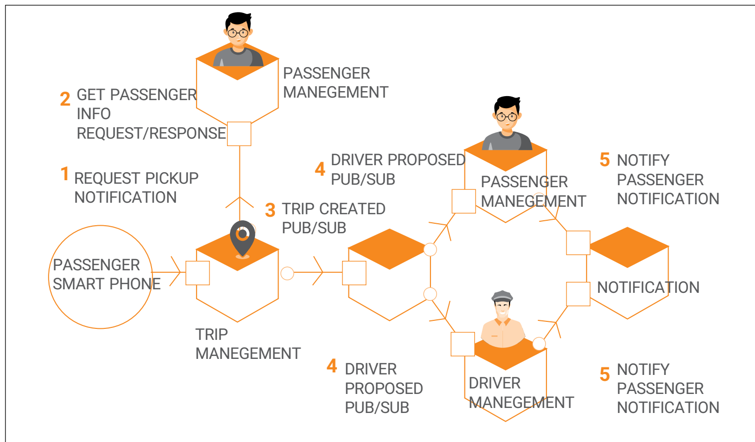
■ **روش Request/Response:** احتمالا یکی از ساده‌ترین و مرسوم‌ترین روش‌های ارتباطی همین روش است. کلاینت درخواستی را به سرویس‌دهنده ارسال می‌کند و منتظر می‌ماند تا پاسخ مناسب از سمت سرویس دهنده ارسال شود. در این روش کلاینت در انتظار پاسخ خواهد ماند و با توجه به اینکه در این بازه زمانی احتمالا Thread جاری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

- **روش دوم Notification یا Request یک طرفه:** کلاینت درخواست یا پیامی را برای یک سرویس خاص ارسال می‌کند اما انتظار پاسخ خاصی را ندارد و یا اینکه اصلا 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 گونه ای طراحی و پیاده‌سازی می‌شود که کلاینت نیازی به پاسخ فوری ندارد و می‌داند دریافت پاسخ ممکن است با تاخیر 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 هرتعداد 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 صرفا 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 ماموریت‌کاری خود و رسیدن به هدف نهایی که ثبت درخواست تاکسی برای یک مسافر است استفاده می‌کنند. از نرم‌افزار موبایل یک Notification برای سرویس Trip Management ارسال می‌شود. سرویس Trip Management با روش Request/Response از وضیعت حساب کاربر مطلع می‌شود. سپس سرویس Trip Management یک Trip ایجاد می‌کند و با روش 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، بیایید با هم نحوه تعریف APIها و نکاتی که باید هنگام طراحی آن‌ها مد نظر داشته‌باشیم 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 گرفته‌است 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 ارائه‌شده اقدام به طراحی و پیاده‌سازی خود سرویس کنیم. طرفداران این روش طراحی به این نکته قائل هستند که طراحی APIها در ابتدا شانس تولید سرویسی که خدمات درستی به کلاینت‌های خود می‌دهد را افزایش می‌دهد.

در ادامه خواهیم دید که انتخاب روش IPC چگونه بر روال 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 تغییر API یک قسمت بسیار ساده‌است و با توجه به اینکه استفاده کننده‌های یک API با خطا مواجه خواهندشد تقریبا می‌توان مطمئن بود با تغییر API تمامی کلاینت‌های آن نیز به روز خواهندشد. 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. بعضا این احتمال وجود دارد که APIهای شما

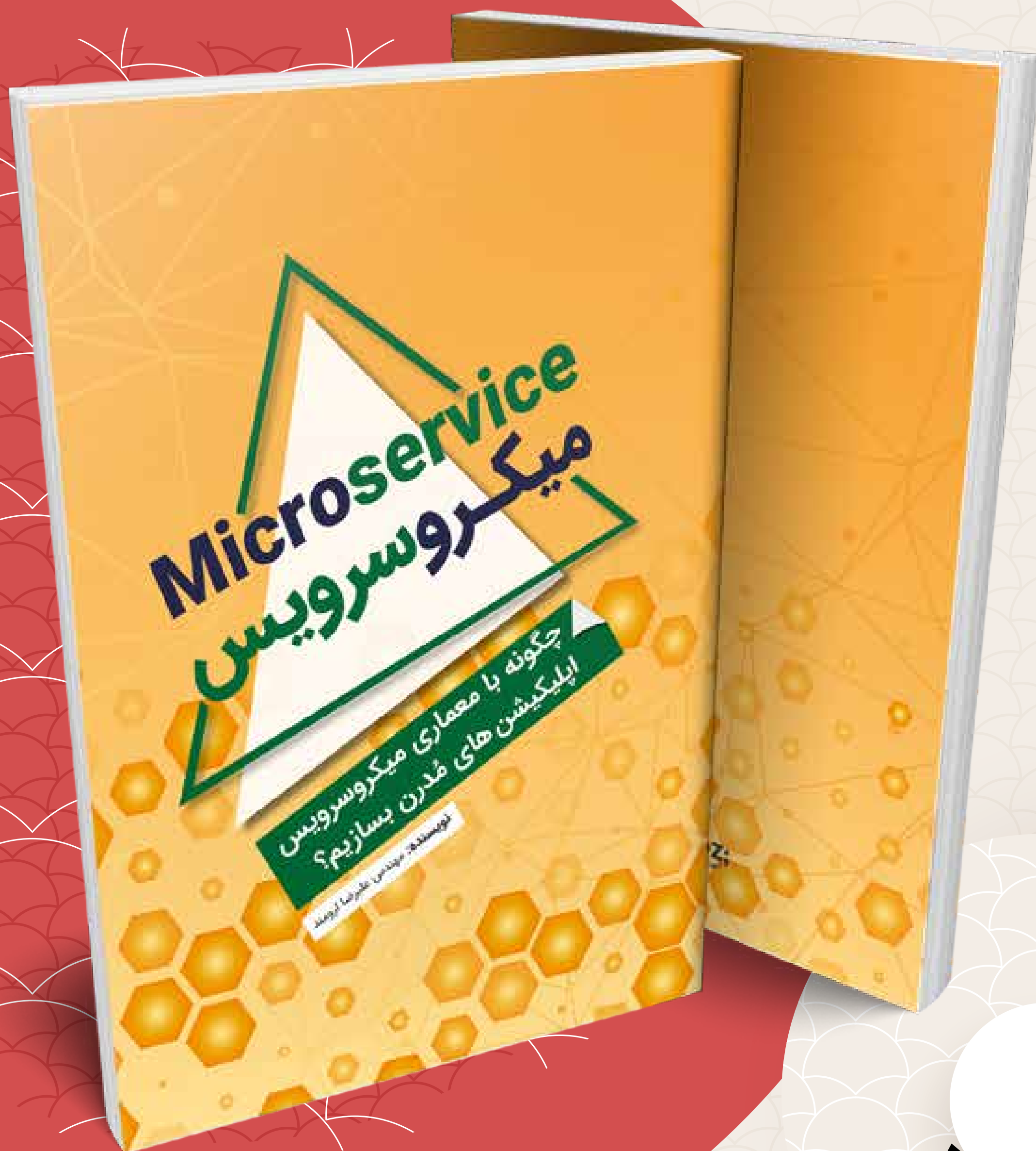
تغییر نکند و صرفا تکامل بیابد و در این شرایط می‌توان توقع داشت که شما کلاینت‌هایی داشته‌باشید که با نسخه‌ها و امکانات قدیمی کار می‌کنند یا کلاینت‌هایی که خود را به روز کرده و از ویژگی‌های جدید APIهای شما استفاده می‌کنند. در هر صورت چه شما تغییر داشته‌باشید و چه تکامل داشتن استراتژی‌هایی برای برخورد با 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 ادامه‌دهند. 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 داشته‌باشند. 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 داشته‌باشید که قابلیت Backward Compatibility را ندارند و به هرحال

برای خرید کلیک کنید.

www.nikamooz.com/mb



کتاب میکروسرویس

نویسنده: مهندس علیرضا ارومند