دانشکده فنی مهندسی

مهندسی کامپیوتر

پایان‌نامه کارشناسی

بررسی فناوری ها در توسعه‌ی برنامه های فراوان‌ـ داده و پیاده سازی نمونه
(Microservice, Docker , GraphQL)

محمدمهرداد شهیدی

استاد راهنما

دکتر مهدی سخایی‌نیا

بهمن ۱۴۰۰

تقدیم به

پدر و مادرم

که بدون زحمات ایشان هیچکدام از این ها ممکن نبود

تشکر از
استاد گرامی دکتر مهدی سخایی‌نیا
بابت کمک های همیشگی ایشان

چکیده

کلمات کلیدی

Data-intensive Apps, GraphQL, Docker and Container, Microservice, NoSQL, CI/CD

فهرست مطالب

فصل ۱

برنامه‌های فراوان‌ـ داده [۱]

۱.۱ مقدمه

در چند دهه‌ی گذشته توسعه های بسیاری در سیستم های پایگاه داده [۲] ، سیستم های توزیع شده [۳] و نهایتا اپلیکیشن هایی که از آن ها استفاده میکردند، صورت گرفته است. [۳]

عوامل زیادی در این توسعه ها نقش داشتند، تعدادی از آنها شامل:

- شرکت های بزرگ اینترنتی نظیر Google، Facebook، Yahoo، Amazon، Microsoftو Twitter که با حجم زیاد داده و ترافیک سروکار دارند، برای اینکه بتوانند به طور کارآمد در این مقیاس کار کنند مجبور به ساخت ابزارهای جدید شدند.
- نیاز کسب کارها به چابک بودن، تست کم هزینه ي فرضیات، پاسخگویی سریع به تغییرات در بازار با کوتاه کردن چرخه‌های توسعه‌ی [۴] خود و مدل های داده ای انعطاف پذیر.
- فرکانس پردازنده ها پیشرفت چشم گیری نداشته اما پردازنده های چند هسته ای بیشتر استفاده می شود و شبکه های کامپیوتری سریع تر شده اند و این به معنی این است که موازی سازی [۵] همواره در حال توسعه خواهد بود.
- تیم های کوچک نرم‌افزاری نیز با استفاده از زیرساخت هایی که به عنوان سرویس (IaaS)[۶] توسط شرکت های بزرگ فراهم می شود (مانند AWS[۷]) می‌توانند سیستم هایی بسازند که در سراسر دنیا توزیع شده است.
- این روز ها از سرویس ها انتظار می رود همواره در دسترس باشند و مدت زمان از کار افتادگی طولانی به منظور نگه داری، قابل قبول نمی باشد.

برنامه های فراوان‌ـ داده همواره مرز این فناوری ها را به جلو تر برده است. منظور از برنامه های فراوان‌ـ داده، برنامه هایی هستند که چالش اصلی آنها داده ها باشند؛ مثلا حجم داده، پیچیدگی داده و یا سرعت تغییر داده چالش ما باشد، در مقابل برنامه های پر محاسبه [۸] هستند که پردازنده گلوگاه [۹] برنامه هستند.

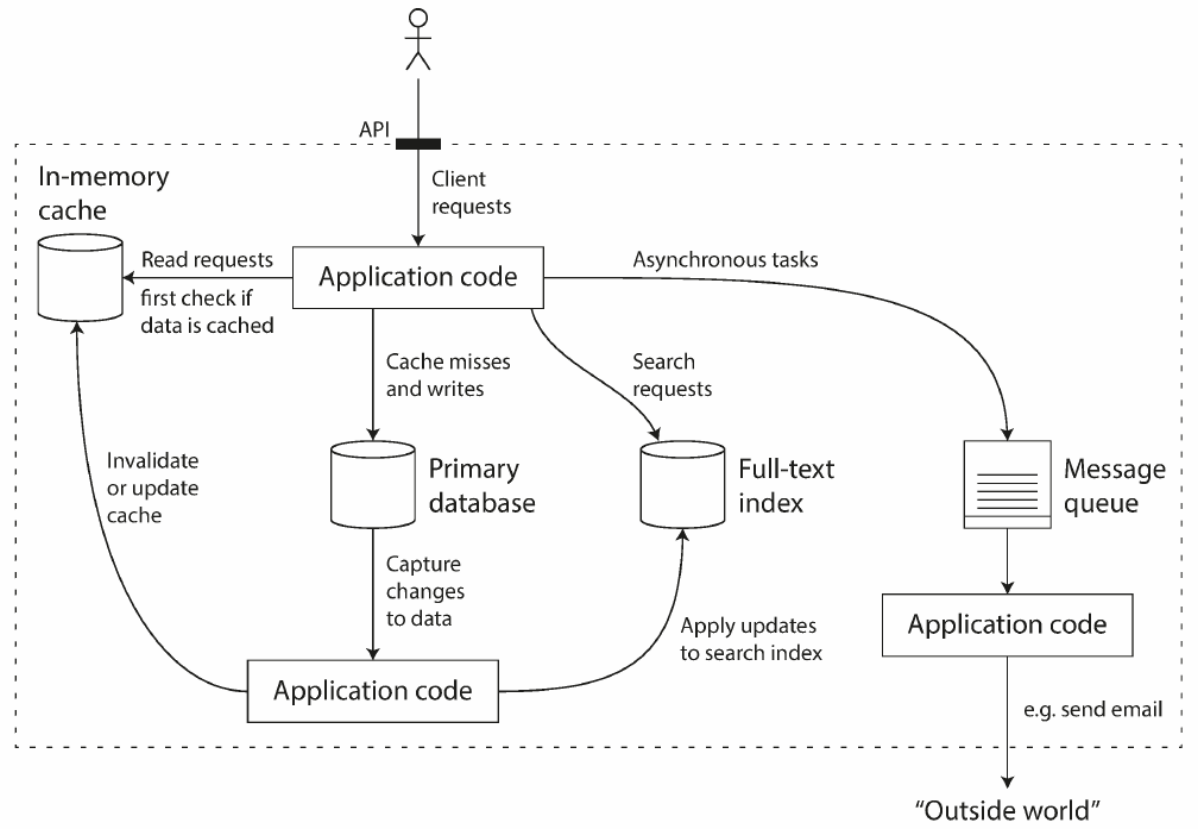
---

[۱] Data-intensive Application
[۲] Database
[۳] Distributed systems
[۴] development cycles
[۵] Parallealism
[۶] Infrastructure as a service
[۷] Amazon Web Services
[۸] compute-intensive
[۹] Bottleneck

۲.۱ عناصر سازنده‌ی معمول در برنامه های فراوان‌ـ داده

برنامه های فراوان‌ـ داده معمولا از عناصر سازنده‌ی استانداردی ساخته می شوند که علمکردهایی برای نیاز های متدوال برنامه ها فراهم می کنند. برای مثال، برنامه ها نیاز هایی از قبیل:

- ذخیره داده ها به منظور بازیابی دوباره ی آن ها **(Database)**
- به یاد داشتن نتیجه ی عمل های پر هزینه ، برای سرعت دادن در خواندن ها **(Cache)**
- اجازه‌ی جستجو به کاربر با واژه‌های کلیدی و یا فیلتر کردن به روش های مختلف **(Search indexes)**
- ارسال پیام به فرآیند دیگر، به منظور پردازش های غیر همزمان [۱۰] **(Stream processing)**
- پردازش دوره‌ای داده های انباشته شده **(Batch processing)**



شکل ۱.۱: یک نمونه معماری برای یک سیستم داده‌ای

زمانی که شروع به توسعه‌ی نرم‌افزار می‌کنیم ،توسعه دهندگان معمولا از ابتدا شروع به ساخت این عناصر برای ذخیره یا پردازش داده نمی‌کنند و از پایگاه‌داده های از قبل ساخته شده استفاده می‌کنند. با این وجود انتخاب یک ابزار و فناوری به سادگی اتفاق نمی افتد؛ چرا که به عنوان مثال سیستم های پایگاه‌داده ای زیادی با ویژگی های مختلف وجود دارند که هر کدام به منظور استفاده‌های خاص منظوره‌ای ایجاد شده اند و همینطور راه‌های متنوعی برای Cache و ایجاد Search indexes وجود دارد.

در سال‌های اخیر ابزارهای بسیاری برای ذخیره و پردازش داده ایجاد شده‌اند، که هر کدام برای استفاده های متفاوتی بهینه شده‌اند و دیگر با دسته بندی های مرسوم سیستم‌های پایگاه‌داده‌ای سازگار نیستند. از طرفی دیگر نرم‌افزار های امروزی نیازمندی های گسترده و متنوعی دارند که یک ابزار به تنهایی جوابگوی نیازهای آن ها نیست.

[۱۰] Asynchronous

۳.۱ فاکتور های تاثیرگذار در برنامه‌های فراوان_ داده

برای طراحی یک سیستم داده‌ای یا سرویس، سوال‌های بسیاری وجود دارد. چطور سیستم تضمین می‌کند که داده تماماً و صحیح در سیستم می ماند، حتی اگر مشکلی در قسمتی از سیستم ایجاد شود؟ چطوری می توانید یک کارایی متداوم برای کاربر فراهم کنیم، حتی اگر در بخشی از سیستم اختلالی پیش آید؟ در صورتی که بارداده‌ای افزایش یافت، چطور سیستم را مقیاس می‌دهیم؟ یک API[۱۱] مناسب برای سرویس برنامه به چه شکل خواهد بود؟

سه فاکتور اصلی که در طراحی و توسعه‌ی اکثر برنامه‌های فراوان_ داده تاثیر دارند:

۱. قابلیت اطمینان [۱۲]
۲. مقیاس‌پذیری [۱۳]
۳. قابلیت‌نگه داری [۱۴]

۱.۳.۱ قابلیت اطمینان

منظور از نرم‌افزاری که قابلیت اطمینان داشته باشد:

- نرم‌افزار کاری را انجام دهد که کاربر از آن انتظار دارد
- قابلیت تحمل اشتباه کاربر یا استفاده‌ی غیر منتظره از نرم‌افزار را داشته باشد
- نرم‌افزار برای کاری که طراحی شده، درصورت بار و حجم داده‌ای مورد انتظار، کارایی خود را حفظ کند
- سیستم از سوء استفاده و دسترسی‌های غیرمجاز جلوگیری کند

با این وجود خطاهایی ممکن است در سیستم اتفاق بیافتد. به سیستمی که بتواند این خطاهارا پیش‌بینی کند و از عهده‌ی آن‌ها برآید **سیستم تحمل‌پذیرخطا** [۱۵] یا **ارتجاعی** [۱۶] می‌گوییم. باید به این نکته توجه کنیم که خطا متفاوت از **شکست** [۱۷] است. خطا به منظور انحراف جزئی سیستم از وظایف خود است. و شکست به معنی این است که سیستم به طور کلی نتواند سروسیی که قرار است فراهم کند را به کاربر ارائه دهد.

کاهش دادن احتمال خطا به صفر یک عمل غیر ممکن است؛ بنابراین تلاش بر این است که سیستم تحمل پذیر به خطا، طوری طراحی شود که از وقوع خطاهایی که منجر به شکست می‌شوند جلوگیری کنیم.

در این سیستم ها می توانیم خطاهایی به طور عمد و حساب شده در سیستم ایجاد کنیم و توانایی سیستم در تحمل پذیری این خطاها را همواره تست و ارزیابی کنیم، و این اطمینان را ایجاد کنیم که سیستم در صورت بروز خطا به طور طبیعی به شکست منتهی نمی شود.

خطاهای سخت‌افزار

خطاهایی که در سطح سخت افزار ممکن است رخ دهد به عنوان مثال: کرش کردن هارددیسک، ازکارافتادگی RAM، خاموشی شبکه برق، یا قطع شدن سیم‌ها در شبکه و...

---

[۱۱] Application Programming Interface
[۱۲] Reliability
[۱۳] Scalability
[۱۴] Maintainability
[۱۵] fault-tolerant
[۱۶] resilient
[۱۷] Failure

هاردیسک هاطبق گزارشات، MTTF[18] در حدود ۱۰ تا ۵۰ سال دارند. بنابرا این در یک دیتاسنتر شامل ۱۰۰۰۰ هارد دیسک در یک خوشه باید انتظار از کار افتادگی یک هاردیسک در روز را داشته باشیم. اولین راه‌حلی که برای این خطا به ذهنمان خطور می‌کند، افزونگی سخت افزار است. این راه‌حل برای خیلی از اپلیکیشن ها همچنان کافی بوده ولی با افزایش تقاضای پردازشی و داده ای اپلیکیشن ها، پیشتر اپلیکشن ها شروع به استفاده از تعداد بیشتر از ماشین ها کردند و این نسبت خطاهای سخت افزاری را به مراتب زیاد تر کرد. این قضیه سیستم ها رو به سمتی برد که توانایی تحمل از دست دادن یک ماشین به طور کلی رو داشته باشند.

خطاهای نرم‌افزار

دسته‌ي دیگر خطاها، خطاهای سیستمی هستند که پیش‌بینی آن‌ها به مراتب سخت تر از خطاهای سخت‌افزاری است و بخاطر وجود وابستگی بین گره ها در سیستم به یک دیگر، علاوه بر شکست خود سیستم، باعث شکست سیستم های وابسته به خود می‌شوند.

برای این دسته از خطا ها معمولا راه‌حل سریعی وجود ندارد ولی کارهای کوچیک زیادی می توانند به ما کمک کنند: با دقت بررسی کردن فرضیات و تعاملات در سیستم؛ از طریق تست کردن؛ ایزوله کردن process؛ اجاز کرش و ریستارت process؛ سنجیدن؛ مانیتور کردن؛ آنالیز کردن رفتار سیستم در عمل

خطاهای انسان

سیستم ها طراحی می شوند تا توسط انسان استفاده شوند و انسان یکی از عواملی هستند که می توانند در سیستم خطا ایجاد کنند. خب چطوری می توانیم یک سیستم داشته باشیم که قابلیت اطمینان خود را با وجود تعامل انسان با آن حفظ کند. برای حل این قضیه چند رویکرد می توانند کمک کننده باشند:

- سیستم را طوری طراحی کنیم که فرصت برای خطا را به حداقل برساند. برای مثال API ای که نحوه ی کار مشخص و ثابتی دارد.
- جداسازی [19] قسمت هایی که خطا ممکن است منجر به شکست سیستم شود از قسمت هایی که بیشترین خطاها را می دهد.
- انجام تست ها در تمام مراحل سیستم، شامل: تست های واحد [20]، تست های ادغامی سراسر سیستم [21] و تست های دستی [22]. همچنین اتوماتیک کردن تست ها به ما کمک می کند تا خطاهایی که به صورت کمیاب در سیستم رخ می‌دهند را نیز پوشش دهیم.
- قابلیت بهبود سریع و راحت سیستم در صورت بروز خطاهای انسانی. برای مثال قابلیت برگشت به عقب تنظیمات یا اضافه کردن تغییرات به صورت تدریجی (کمک کردن به شناسایی باگ های غیر منتظره)
- ایجاد سیستم مانیتورینگ شفاف و با جزئیات، شامل معیار های عملکردی سیستم و نرخ بروز خطا.

قابلیت اطمینان چقدر اهمیت دارد؟

قابلیت اطمینان فقط برای مراکز انرژی هسته ای و یا کنترل ترافیک هوایی اهمیت ندارد مثلا در سیستم هایی که در کسب کار ها وجود دارند نبود اطمینان در سیستم می تواند هزینه های بسیار زیادی چه از نظر مالی و شهرتی برای شرکت ها ایجاد کند.

---

[18] Failure To Time Mean
[19] decouple
[20] unit test
[21] whole-system integration test
[22] manual test

شرایطی وجود دارد که ممکن است برای کاهش هزینه ی توسعه یا عملیاتی کردن سیستم بخواهیم از قابلیت اطمینان تا حدی چشم پوشی کنیم. ولی همیشه باید در نظر داشته باشیم که انجام این کار چه هزینه هایی ممکن است در آینده برای ما ایجاد کند.

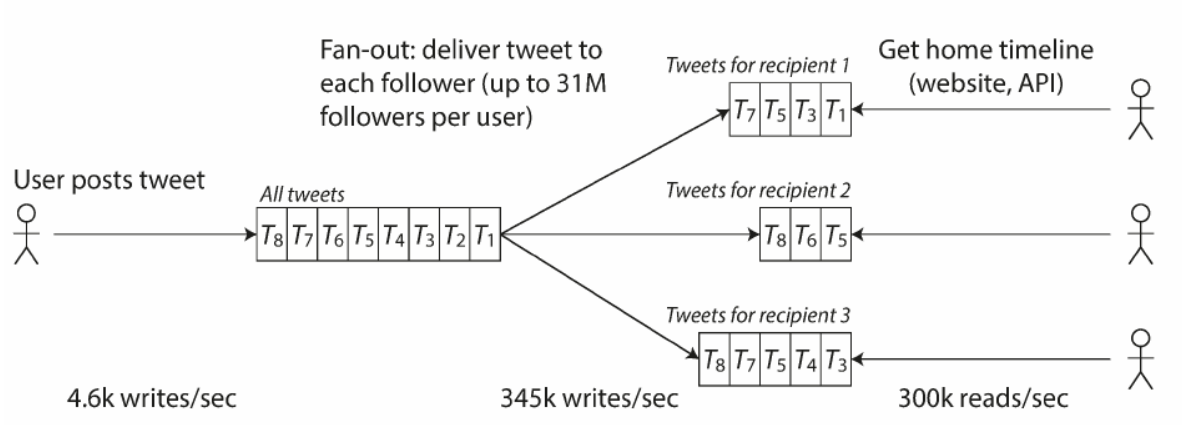
۲.۳.۱ مقیاس پذیری

حتی اگر سیستمی که امروز داریم قابلیت اطمینان بالایی داشته باشد این به این معنی نیست که در آینده نیز این قابلیت را می تواند حفظ کند. این اتفاق می تواند علت های مختلفی داشته باشد، مانند: افزایش کاربران همزمان در سیستم و یا نیاز سیستم به پردازش بار داده ای بیشتر از قبل.

منظور از **مقیاس‌پذیری** قابلیت سیستم برای مقابله با بار افزایش یافته‌ی سیستم است. یا به عبارتی دیگر اگر سیستم در یک میسر خاص رشد کرد، چه انتخاب هایی برای مقابله با این قضیه داریم؟

توصیف بار [۲۳]

برای اینکه مفهوم رشد سیستم را بفهمیم ابتدا باید توصیفی از **بار** در سیستم داشته باشیم. بار را می توان با پارامتر های عددی نشان داد که به آن **پارامتر های بار** [۲۴] گفته می‌شود. انتخاب بهترین پارامتر برای توصیف بار در سیستم به معماری سیستم بستگی دارد. چند نمونه از پارامتر های بار: تعداد درخواست های در ثانیه به یک وب سرور، نسبت نوشتن‌ها به خواندن‌ها در پایگاه داده، تعداد کاربران فعال به طور همزمان در یک چت روم، نزخ اصابت [۲۵] در حافظه‌ی نهان.



شکل ۲.۱: پارامتر های بار توییتر در سال ۲۰۱۲

توصیف عملکرد

بعد از اینکه بار را در سیستم تعریف کردیم می‌توانیم بررسی کنیم که با افزایش بار چه اتفاقی برای سیستم می‌افتد. به این مسئله می‌توان به دو صورت نگاه کرد:

- تاثیرات در عملکرد سیستم هنگام افزایش پارامترهای بار بدون اینکه منابع سیستم(CPU,RAM,...) را تغییر دهیم به چه صورت خواهد بود؟

---

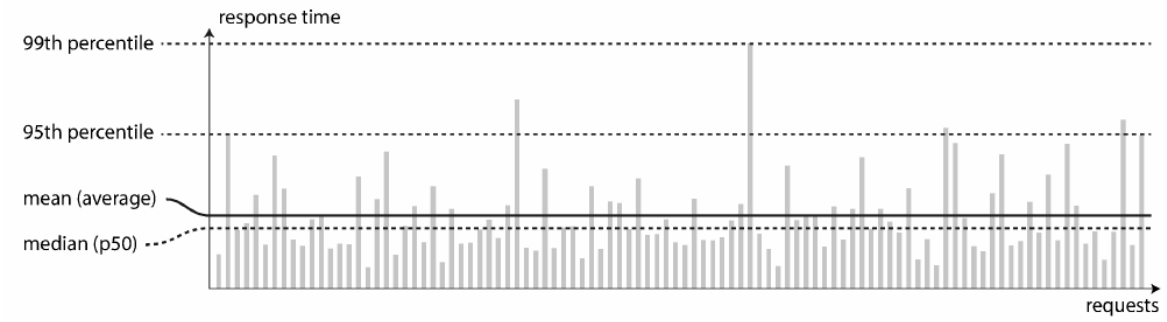
[۲۳] Load
[۲۴] load parameters
[۲۵] hit ratio

- زمانی که پارامترهای بار را افزایش می دهیم، منابع سیستم را چه مقدار باید افزایش دهیم تا عملکرد سیستم بدون تغییر باقی بماند؟

برای جواب به هردو سوال به یک سری از پارامتر های عددی برای نشان دادن عملکرد سیستم نیاز داریم.

برای مثال در سیستم های batch processing مانند Hadoop، توان عملیاتی [۲۶]- تعداد پردازش record در ثانیه یا زمان کلی که برای انجام یک عمل در یک dataset به سایز معین انجام می‌شود برای ما اهمیت دارد. ولی در سیستم های آنلاین چیزی که بیشتر برای ما حائز اهمیت است، **زمان پاسخگویی** [۲۷] سرویس ها هست - زمان بین ارسال یک درخواست توسط کاربر و دریافت پاسخ آن درخواست.

زمان پاسخگویی برای درخواست حتی اگر یک درخواست را چند بار ارسال کنیم، همواره اندکی متفاوت از قبل خواهد بود. و این تفاوت در سیستم‌ها در عمل که درخواست های متنوعی را جواب می دهند بیشتر خواهد بود، به همین دلیل زمان پاسخگویی برای ما یک عدد نیست بلکه **توزیعی** [۲۸] از این مقدارها است که می توانیم آن‌ها را بسنجیم.(شکل ۳.۱)



شکل ۳.۱: زمان های پاسخگویی برای یک نمونه‌ی ۱۰۰ تایی درخواست ها برای یک سرویس.

روش‌های مقابله با بار داده‌ای

خب حالا بعد از اینکه پارامترهای بار و معیارهای سنجش عملکرد را توصیف کردیم، این سوال پیش میاد که چطور می‌توانیم یک عملکرد خوب برای سیستم داشته باشیم حتی اگر افزایش در پارامتر های بار در سیستم داشته باشیم.

یک معماری که برای یک بار معین مناسب است، بعید است که بتواند ۱۰ برابر همان بار را در سیستم تحمل کند. پس اگر یک سرویس با سرعت رشد بالا دارید، احتمالا باید برای هر مرتبه از بار، در معماری خود تجدید نظر کنید.

معمولا در مقیاس دادن سیستم دوگانگی وجود دارد که این مقیاس پذیری به صورت **عمودی** [۲۹] باشد یا به صورت **افقی** [۳۰] . ولی در عمل ترکیبی از این دو می‌تواند معماری بهتری باشد. برخی از سیستم ها نیز می‌توانند به صورت ارتجاعی [۳۱] باشند: سیستم با شناسایی افزایش بار منابع محاسباتی خود را افزایش می‌دهد. در مقابل سیستم می‌تواند به صورت دستی [۳۲] مقیاس بپذیرد. سیستم های ارتجاعی برای شرایطی که بار سیستم غیرقابل پیش بینی هست مناسب است ولی مقایس دهی دستی به مراتب ساده تر و غافلگیری کمتری در عملیاتی کردن وجود دارد.

[۲۶] throughput
[۲۷] response time
[۲۸] distribution
[۲۹] vertical scaling
[۳۰] horizontal scaling
[۳۱] elastic
[۳۲] manual

توزیع کردن یک سرویس که به صورت stateless هست به مراتب ساده‌تر و سرراست تر از سرویسی هست که stateful هست. به همین علت معمولا برای سرویس های stateful ترجیحا از مقیاس دهی عمودی استفاده می‌شد تا زمانی که مجبور باشیم به علت بار زیاد به مقیاس دهی افقی روی بیاوریم.

یک معماری که برای یک برنامه به خوبی مقیاس می‌پذیرد با این فرضیات که کدام عملیات در سیستم متدوال و کدام متداول نیست ساخته شده است(پرامترهای بار). با وجود اینکه معماری مقیاس پذیر برای یک برنامه‌ي خاص ممکن است ساخته بشود ولی معمولا از یک سری عناصر سازنده‌ی متداول ساخته شده اند و دارای الگوی های مشخصی هستند.

۳.۳.۱ قابلیت نگه‌داری

کاملا واضح است که بیشترین هزنیه یک نرم‌افزار برای توسعه اولیه آن نیست بلکه برای نگه داری آن است. از جمله: رفع باگ‌ها، قابل استفاده نگه داشتن سیستم، بررسی شکست ها، سازگار کردن آن با پلتفرم‌های جدید، تغییر دادن آن برای نیازهای جدید، پرداختن هزینه‌های فنی، اضافه کردن قابلیت های جدید.

خیلی از کسانی که روی سیستم‌های نرم‌افزاری کار می‌کنند خیلی تمایلی به نگه‌داری سیستم های قدیمی [۳۳] ندارد؛ به دلایل مختلف از قبیل: رفع مشکلاتی که توسط توسعه‌دهندگان قبلی ایجاد شده، کار کردن بر روی پلتفرمی که منسوخ شده، یا سیستمی که مجبور شده کار هایی انجام دهد که برای آن طراحی نشده است.

با این وجود، ما می‌تونیم نرم‌افزارمان را طوری طراحی کنیم که مشکلات نگه‌داری را تا حد زیادی کاهش دهیم . سه تا از اصولی که به ما در طراحی سیستم‌های نرم‌افزاری کمک می‌کنند

- قابلیت عملیاتی شدن[۳۴]
- سادگی[۳۵]
- تکامل‌پذیری[۳۶]

مانند قابلیت‌اطمینان و مقیاس‌پذیری یک راه‌حل ساده برای اینکه سیستم را قابل نگه‌داری کنیم وجود ندارد ولی در ذهن داشتن این اصول در طراحی به ما کمک می‌کند.

قابلیت عملیاتی شدن

ساده کردن عملیاتی کردن سیستم به منظور اینکه تیم عملیات بتواند سیستم را به طور متداوم در اجرا نگه دارد. یک تیم عملیات خوب معمولا مسئولیت هایی از قبیل:

- مانیتور کردن سلامت سیستم و بازگردانی سریع سرویس هایی که به وضعیت نامناسب می روند.
- رهگیری علل مشکلات پیش‌آمده در سیستم مانند: شکست سیستم یا کاهش عملکردی آن.
- به روز نگه‌داشتن سیستم و پچ کردن سیستم از جمله پچ های امنیتی
- پیش‌بینی مشکلات احتمالی در سیستم و حل کردن آن‌ها قبل از وقوع.
- ایجاد ابزار و راهکارهای مناسب برای پیاده‌سازی، مدیریت تنظیمات و غیره ...
- انجام نگه‌داری های پیچیده مانند انتقال برنامه از یک پلتفرم به پلتفرم دیگر.
- امن نگه داشتن سیستم در هنگام تغییرات در تنظیمات

---

[۳۳] legacy system
[۳۴] Operability
[۳۵] Simplicity
[۳۶] Evolvability

- ایجاد پروسه هایی که عملیات ها را قابل پیش‌بینی می‌کند و به پایدار نگه‌داشتن محیط تولید کمک می‌کند
- نگه‌داری دانش سازمان درباره‌ی سیستم، حتی موقعی که افراد از تیم خارج یا وارد می‌شوند.

منظور از یک سیستم با قابلیت‌عملیاتی خوب سیستمی است که کارهای روتین عملیات را ساده کند و این اجازه را به تیم عملیات بدهد که تلاش خود را به سمت فعالیت های باارزش تری ببرند. از جمله :

- فراهم کردن قابلیت رویت [۳۷] برای رفتار در حال اجرای سیستم و مانیتورینگ خوب برای اجزای داخلی سیستم
- فراهم کردن پشتیبانی مناسب برای اتوماسیتون و تلفیق [۳۸] با ابزار استاندارد
- اجتناب از وابستگی به دستگاه های منفرد(کل سیستم کار کند حتی اگر یک دستگاه از سیستم از کار بیافتد.)
- فراهم کردن مستندات کامل درباره‌ی سیستم و یک مدل عملیاتی قابل فهم راحت
- فراهم کردن رفتار پیش‌فرض خوب برای سیستم و همینطور قابلیت تغییر راحت رفتار پیش‌فرض
- قابلیت خود ترمیمی، و هیمنطور فراهم کردن کنترل برای ادمین سیستم برای تغییر وضعیت سیستم
- قابل‌پیش بینی کردن رفتار و کاهش غافلگیری ها در سیستم

سادگی

پروژه های کوچک می‌توانند خیلی ساده و کدی گویا داشته باشند. ولی از زمانی که پروژه ها شروع به بزرگ تر شدن می‌کنند، پیچیده تر می‌شوند و فهم آن‌ها دشوار تر می‌شود. این پیچیدگی باعث کاهش سرعت افرادی که روی سیستم کار می‌کنند می‌شود و به مراتب هزینه‌های نگه‌داری را افزایش می‌دهد.

علائم محتمل متنوعی برای پیچیدگی وجود دارد: رشد سریع فضای حالت در سیستم، همبستگی [۳۹] زیاد ماژول ها، وابستگی‌های چندگانه، ناسازگاری در اسامی و اصطلاحات فنی، هک برای افزایش مسئله‌ی عملکردی سیستم و ....

وقتی پیچیدگی نگه‌داری را دشوار می‌کند، معمولا بودجه و زمان‌بندی های فراتراز پیش‌بینی‌ها می‌رود. در نرم‌افزار های پیچیده امکان ایجاد باگ‌ها هنگام تغییر سیستم بیشتر می‌شود. فهم سیستم برای توسعه‌دهندگان سخت‌تر و فرضیات پنهان، عواقب ناخواسته و تعامل های غیرمنتظره ساده تر چشم پوشی می‌شوند.

سادگی به معنی این نیست که کارایی سیستم را کاهش دهیم بلکه تا حد امکان از **پیچیدگی‌های تصادفی** [۴۰] جلوگیری کنیم. پیچیدگی‌هایی که در ذات مسئله ایی که سیستم قرار است حل کند وجود ندارد و فقط در پیاده‌سازی به‌وجود می‌آیند.

یکی از راهکارها برای مقابله با پیچیدگی‌های تصادفی استفاده از انتزاع [۴۱] است که خیلی از جزئیات پیاده‌سازی را می‌تواند برای ما پنهان کند. با این وجود انتخاب یک انتزاع خوب همواره یک کار دشوار است. در حوزه ی سیستم‌های توزیع شده این مسئله خیلی راه حل ثابتی ندارد.

تکامل‌پذیری

احتمال اینکه نیازمندی‌های سیستم برای همیشه ثابت بماند به شدت پایین است. و این قضیه می تواند دلایلی از قبیل: جمع‌اوری واقعیت‌های جدید، مورداستفاده‌های پیش‌بینی نشده، تغییرت اولویت‌ها در کسب‌وکار، درخواست‌های

---

[۳۷] visibility
[۳۸] integration
[۳۹] coupling
[۴۰] accidental complexity
[۴۱] abstraction

جدید از کاربر، جابه جایی از پلتفرم قدیمی به پلتفرم جدید، آیین‌نامه‌های جدید در قوانین، رشد سریع سیستم ونیاز به تغییر در معماری سیستم.

از دیدگاه پروسه‌های سازمانی، الگوی کاری چابک [۴۲] یک چارچوب برای تطابق سریع با تغییرات فراهم می‌کند. همچنین اجتماع چابک ابزار فنی و الگوهای مناسبی برای توسعه‌ی نرم‌افزارها در محیط‌هایی با تغییر مکرر مانند: TDD(Test Driven Developement) توسعه داده است.

امکان اینکه بتوان به راحتی در یک سیستم داده‌ای تغییر ایجاد کنیم و یا سیستم را با تغییرات سازگار کنیم، به طور نزدیکی به سادگی و انتزاع خوب در سیستم بستگی دارد. از آن جایی که این مسئله برای ما اهمیت زیادی دارد، ما از چابک بودن در سیستم‌های داده‌ای با عنوان تکامل‌پذیری یاد می‌کنیم.

۴.۱ خلاصه

در این فصل سعی کردیم تعریف مناسبی از دسته برنامه‌هایی که با عنوان **برنامه‌های فراوان_داده** یاد کردیم، داشته باشیم. به چالش‌هایی که در این برنامه‌ها وجود دارد اشاره داشتیم و عناصر سازنده‌ی معمول این برنامه ها را ذکر کردیم.

سه مورد از فاکتورهای تاثیرگذار در برنامه‌های فراوان_داده که برای ما اهمیت ویژه‌ای دارند و با عنوان نیازمندی‌های non-functional در توسعه نرم‌افزار می‌شناسیم را، بررسی کردیم.

قابلیت اطمینان به منظور اینکه سیستم بتواند به کار خود ادامه دهد حتی اگر خطایی در آن رخ دهد. خطا می‌تواند در سخت افزار یا نرم‌افزار رخ دهد و یا توسط انسان ایجاد شود. سیستمی که قابلیت تحمل‌پذیری خطا را داشته باشد، می‌تواند خیلی از این دسته خطا ها را از دید کاربران سیستم، پنهان کند.

مقیاس‌پذیری به منظور داشتن استراتژی هایی که در صورت افزایش بار (load) بر روی سیستم، عملکرد سیستم را در وضعیت خوب حفظ کنیم. برای اینکه درباره‌ی مقیاس پذیری بحث کنیم ابتدا لازم داریم که بتوانیم بار و عملکرد سیستم را به طور کمی تعریف کنیم.

قابلیت نگه‌داری دارای جنبه‌های مختلفی است و بصورت کلی تلاش بر این است که کار با سیستم را برای توسعه دهندگان و تیم عملیاتی ساده تر کند. از جمله استفاده از انتزاع‌های (abstractions) خوب در سیستم به منظور کاهش پیچیدگی سیستم و راحتی ایجاد تغییرات در سیستم. قابلیت عملیاتی‌شدن خوب به منظور قابلیت رویت سلامت سیستم و روش‌های موثر مدیریت سیستم در اجرا.

برای اینکه سیستمی داشته باشیم که همه‌ی این قابلیت‌ها را که در بالا ذکر کردیم داشته باشد، یک راه‌حل ثابت وجود ندارد اما الگوها و اصولی در طراحی وجود دارد که به ما کمک می‌کند . همچنین فناوری‌ها و روش‌هایی در سال‌های اخیر توسعه‌یافته اند که چالش‌های مشترکی که در این دسته از برنامه‌ها وجود دارند را برطرف می‌کنند که در فصل‌های آینده درباره‌ی آن‌ها صحبت می‌کنیم.

[۴۲] Agile

فصل ۲

معماری میکروسرویس

۱.۲ مقدمه

معماری میکروسرویس یکی از معماری هایی است که در سیستم‌های توزیع شده و مقیاس‌پذیر بسیار استفاده می‌شود. این معماری این قابلیت را می‌دهد که تیم‌های مختلف به صورت مستقل از هم به توسعه‌ی سیستم بپردازند و نگه‌داری و تغییر پذیری سیستم را به مراتب بهبود ببخشد.[۵]

با اینکه معماری میکروسرویس نقاط مثبت زیادی دارد ولی هزینه‌ها و چالش‌هایی در طراحی برای ما ایجاد می‌کند و دربرخی از شرایط توجیه مناسبی برای استفاده از آن نیست. پس باید همواره به این نکته توجه داشت که این معماری یک راه‌حل برای همه‌ی مشکلات نیست و یک سبک‌وسنگین کردن [۱] در انتخاب آن وجود دارد.[۱]

در این فصل **معماری یکپارچه** [۲] را با معماری میکروسرویس مقایسه خواهیم کرد و مختصرا نقاط مثبت و منفی هر کدام از این معماری ها را ارائه می دهیم.

۲.۲ مقایسه میکروسرویس با معماری یکپارچه

معماری های متنوعی در توسعه نرم‌افزار وجود دارد و هر کدام از آن‌ها فواید و عیب هایی دارند و نمی‌توان یک معماری را کامل و مناسب برای همه‌ی شرایط دانست. در این بخش به مقایسه دو معماری میکروسرویس و یکپارچه می‌پردازیم.

۱.۲.۲ فواید معماری یکپارچه

زمانی که نرم‌افزار نسبتا کوچک است و کد بیس بزرگی ندارد معماری یکپارچه فوایدی ازقبیل:

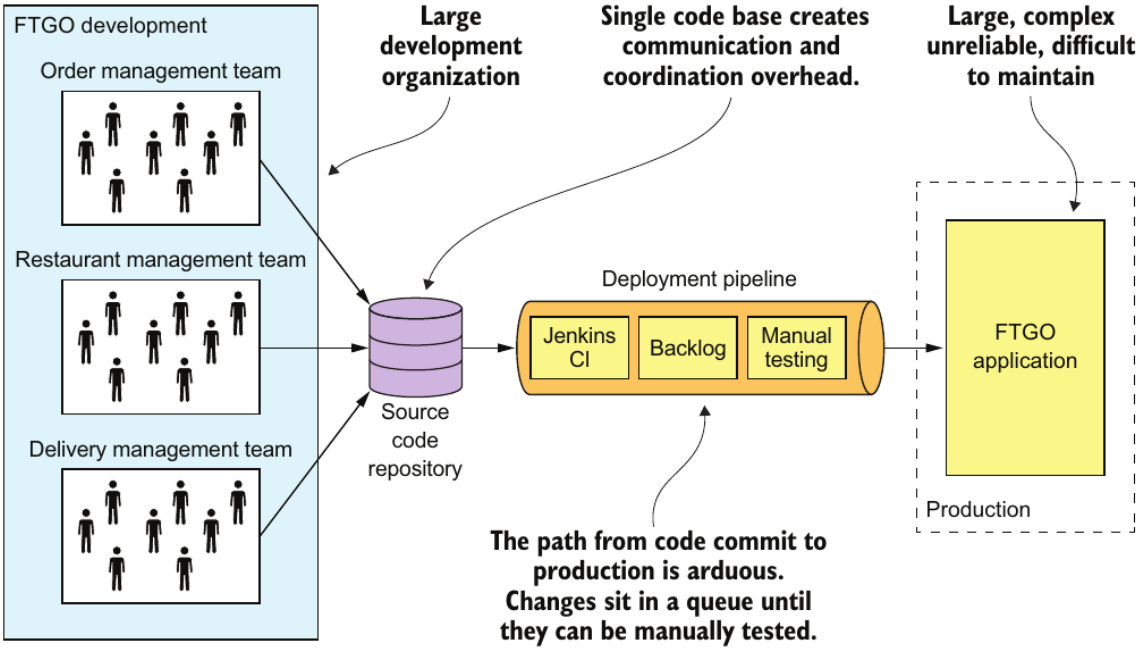
- **توسعه‌ی راحت:** IDE و ابزار نرم‌فزاری معمولا مناسب برای توسعه single application ها هستند.
- **سادگی در انجام تغییرات اساسی در نرم‌افزار:** می‌توانید کد یا schema پایگاه داده را تغییر داده و نتیجه را سریعا پیاده سازی کنیم.
- **تست راحت:** انجام تست های end-to-end برای نرمافزار.
- **سادگی در deploy** یک کد بیس که تمام نرمافزار را شامل می‌شود deploy کردن را به مراتب ساده می‌کند.

---

[۱] trade-off

[۲] monolithic

- **مقیاس‌پذیری راحت** اجرای چند instance از برنامه در پشت یک load balancer.



شکل ۱.۲: یک نمونه از معماری یکپارچه

۲.۲.۲ ایرادهای معماری یکپارچه

زمانی نرم‌افزار شروع به بزرگ تر شدن می‌کند و نیاز به ایجاد تغییرات به صورت دائم در نرم‌افزار باشد، معماری یکپارچه محدودیت‌هایی را ایجاد می‌کند. از جمله‌ی این محدودیت‌ها:

پیچیدگی در نرم‌افزار

یکی از مسائلی که وقتی نرم‌فزار شروع به بزرگ تر شدن می‌کند، پیچیدگی نرم‌افزار است. پیچیدگی فهم نرم‌افزار را برای توسعه‌دهندگان سخت و نهایتا رفع باگ‌ها و اضافه کردن feature ها به نرم‌افزار را سخت و نهایتا غیر ممکن می‌کند.

کند شدن روند توسعه

نرم‌افزار بزرگ توسعه را به مراتب کند می‌کند، ساخت و تست هر تغییر در نرم‌افزار پروسه‌ی طولانی تری می‌شود که این قضیه در کارآمد بودن تیم توسعه تاثیر می‌گذارد.

دشواری در مقیاس‌پذیری

در یک نرم‌افزار، ماژول‌های مختلف آن نیازهای منابعی متفاوتی دارند به عنوان مثال یک ماژول نیاز به CPU بیشتر و ماژول دیگر نیاز به memory بیشتر دارد. این تضاد ها در نیازها، مقیاس‌پذیری یک سیستم یکپارچه را دشوار می‌کند.

چالش اطمینان‌پذیری در سیستم‌های یکپارچه

تست کردن یک نرم‌افزار بزرگ یک پارچه کار بسیار دشواری است و این دشواری در تست نهایتا منجر به ایجاد خطا در تولید می‌شود. یکی دیگر از مشکلات، ایزوله نبودن خطاها به دلیل اینکه همه‌ی ماژول ها در یک process اجرا

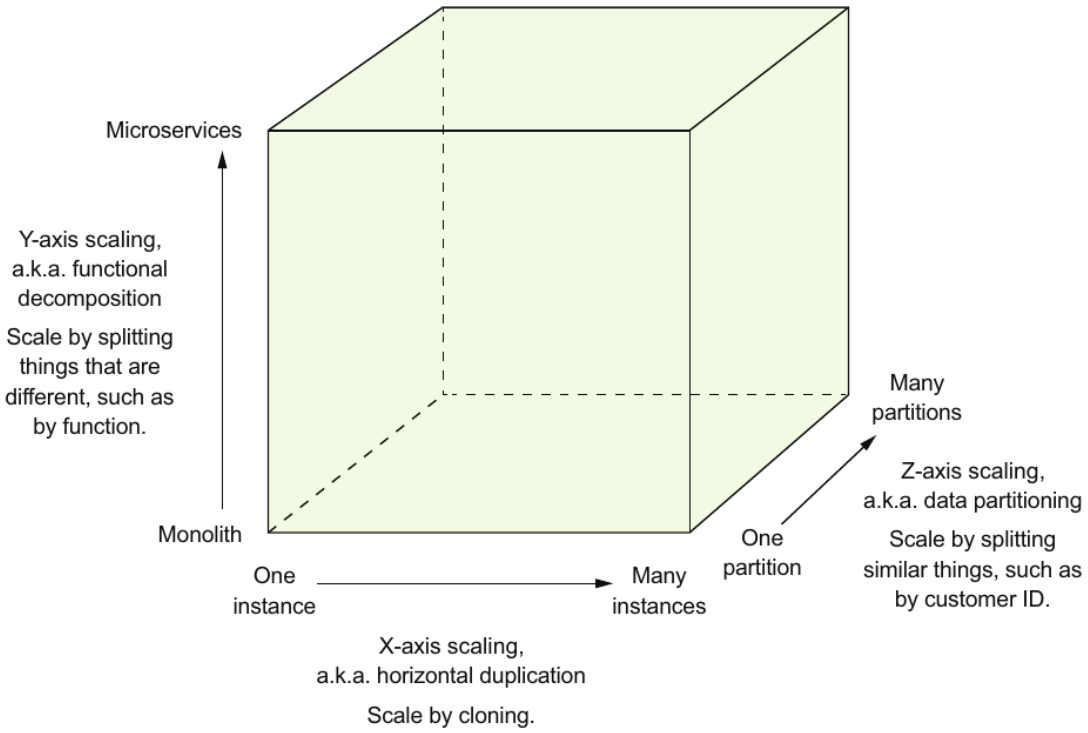
می‌شوند. نهایتا کرش کردن یک ماژول منجر به شکست کامل سیستم می‌شود.

دشواری در تغییر فناوری‌های منسوخ شده در سیستم

تغییر در زبان یا framework در نرم‌افزارهای یکپارچه امکان پذیر نسیت. در نتیجه، توسعه دهندگان انتخاب های محدودی در انتخاب فناوری‌ها دارند ومجبور اند از فناوری‌هایی که در شروع نرم‌افزار انتخاب کردند استفاده کنند.

۳.۲.۲ مکعب مقیاس

برای اینکه مقایسه بهتری بین دو معماری یکپارچه و میکروسرویس داشته باشیم از یک مدل مقیاس‌پذیری سه بعدی در شکل ۲.۲ برای نشان دادن راه‌های مقیاس‌دهی نرم‌افزار استفاده می‌کنیم.



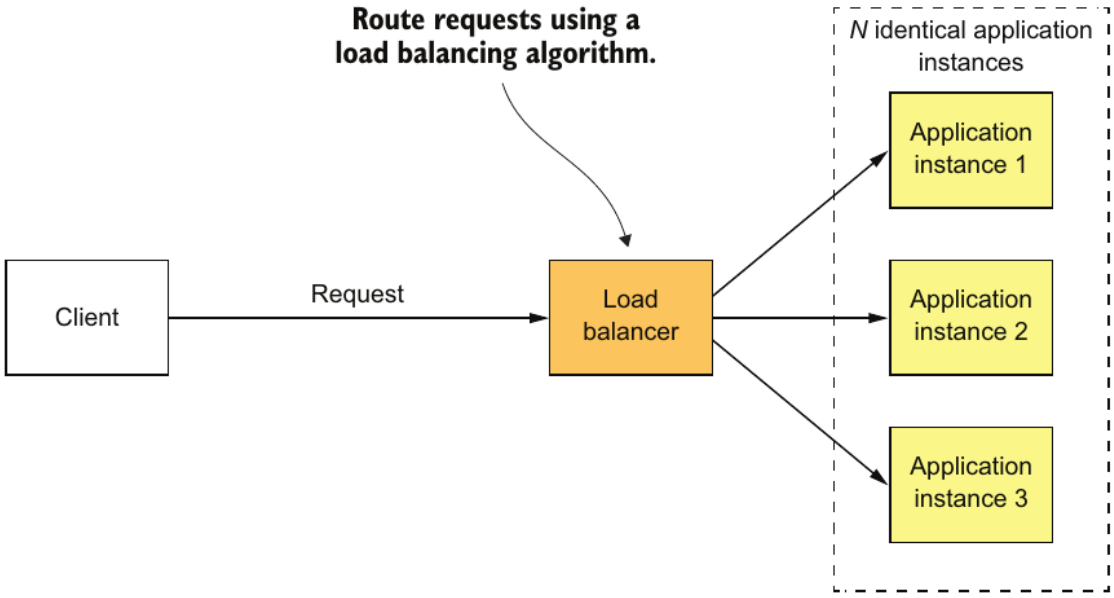
شکل ۲.۲: مدل سه بعدی مقیاس برای نشان دادن راه‌های مختلف مقیاس‌دهی سیستم

محور X متعادل کردن بار در چندین instances

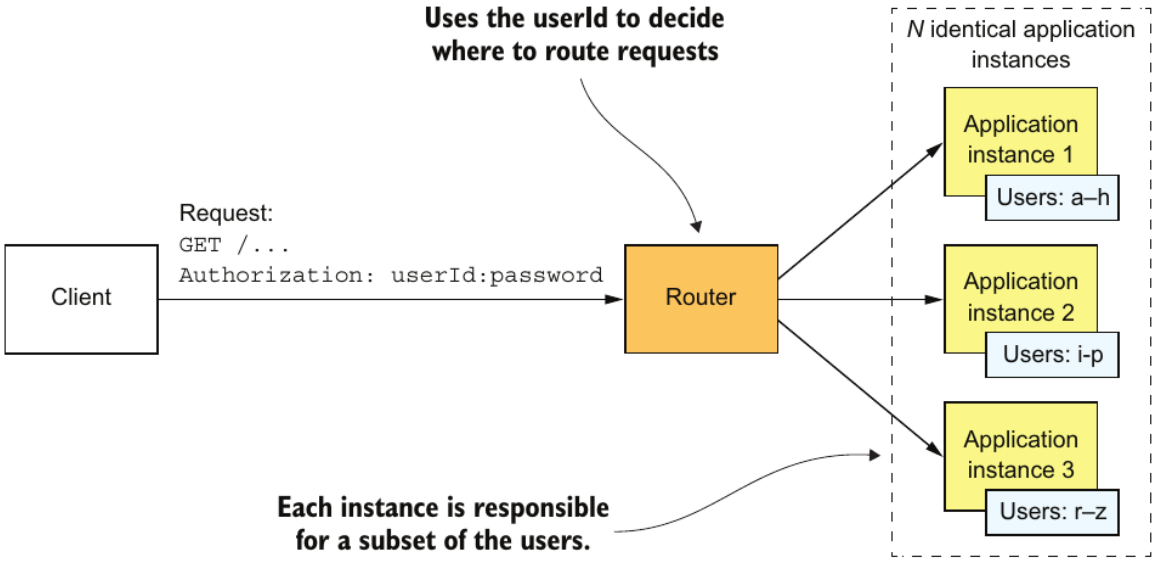
مقیاس‌دهی متدوال در برنامه‌های یکپارچه اضافه کردن چندین instances در پشت یک Load balancer . Load balancer با توزیع درخواست ها بین instance N مشابه، ظرفیت و دسترس‌پذیری برنامه را ارتقا می دهد.(شکل ۳.۲)

محور Z مسیریابی درخواست‌ها بر اساس خصوصیت درخواست‌ها

بر خلاف مقیاس در محور X هر instance مسئولیت بخشی از داده‌ها را به عهده می‌گیرد. مسیریاب بر اساس ویژگی‌های درخواست برای مثال شناسه‌ی کاربر یکی از این instance را معین می‌کند. این مقیاس‌دهی برای برطرف کردن افزایش تراکنش و حجم داده مناسب است.(شکل ۴.۲)



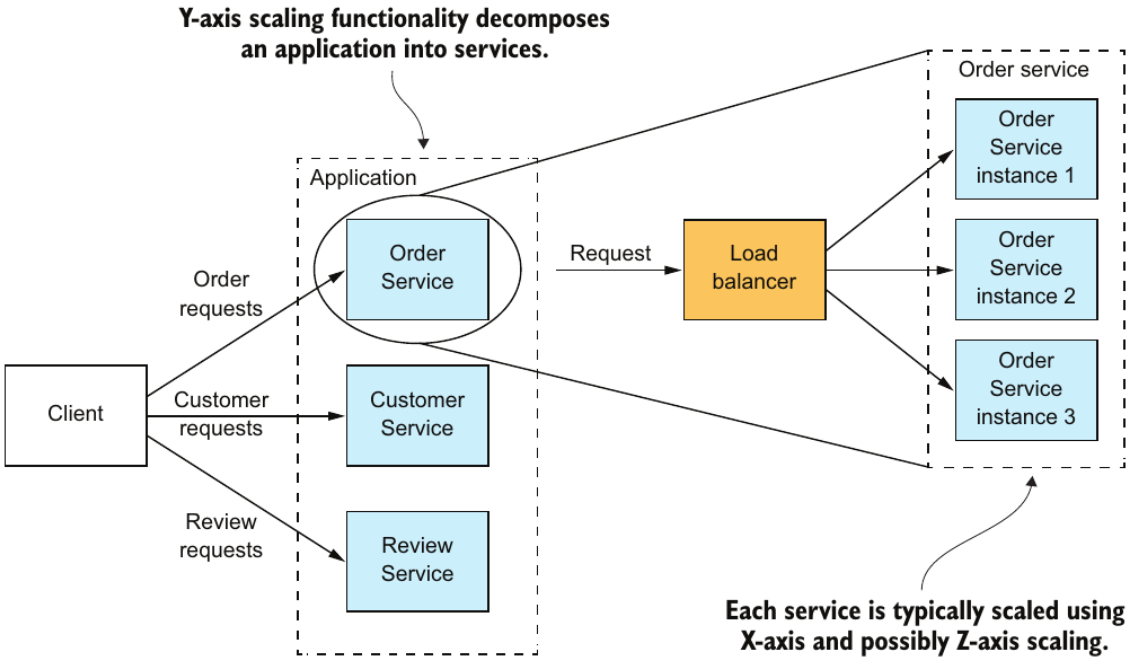
شکل ۳.۲: چندین instance مشابه از یک برنامه یکپارچه در پشت یک load balancer (محور X)



شکل ۴.۲: چندین instance مشابه از یک برنامه یکپارچه در پشت یک مسیریاب. هر instance مسئولیت قسمتی از داده‌ها را به عهده دارد. (محور Z)

محور Y تجزیه‌ی عملکردی برنامه به سرویس‌ها

با وجود اینکه مقیاس‌دهی در هر دو محور X, Z ظرفیت و قابلیت دسترسی برنامه را ارتقا می‌دهند ولی هیچ کدام از آنها مشکل افزایش توسعه و پیچدگی نرم‌افزار را حل نمی‌کنند. برای حل این مشکل از مقیاس‌دهی در جهت محور Y استفاده می‌کنیم. با شکستن یک برنامه‌ی یکپارچه به مجموعه‌ای از سرویس ها. سرویس، مینی برنامه‌ای هست که فقط بروی یک عملکرد و مسئولیت منسجم تمرکز می‌کند. (شکل ۵.۲)



شکل ۵.۲: شکستن برنامه به مجموعه‌ای از سرویس‌ها. (محور Y)

۴.۲.۲ میکروسرویس

فواید:

- بهبود دادن پروسه‌ی تحویل دائم برنامه و پیاده سازی برنامه‌های حجیم و پیچیده
- سرویس های کوچک و قابل نگه‌داری
- deploy کردن مستقل سیستم از یک دیگر
- مقیاس پذیری مستقل سرویس‌ها
- خودمختاری تیم‌های توسعه‌ی هر سرویس
- راحتی آزمایش و به کارگیری فناوری‌های جدید
- ایزوله کردن خطا ها در سیستم

ایرادها:

- شکستن برنامه به مجموعه‌ای از سرویس‌ها همیشه کار ساده‌ای نیست و چالش‌های به همراه دارد
- سیستم‌های توزیع شده پیچیده هستند و که کار توسعه، تست و پیاده‌سازی را دشوار می کند
- هماهنگ سازی سرویس‌ها برای پیاده سازی امکانات مشترک بین آن‌ها پیچیدگی‌هایی ایجاد می کند
- انتخاب اینکه چه زمان باید از معماری میکروسرویس استفاده کرد دشوار است.

۳.۲ خلاصه

در این فصل به صورت مختصر به تاثیر انتخاب معماری مناسب برای نرم‌افزار پرداختیم. **معماری میکروسرویس** که یکی از معماری‌های محبوب برای سیستم‌های توزیع شده و مقیاس پذیر است را با **معماری یکپارچه** مقایسه کردیم.

معماری یکپارچه علارغم خوبی هایی که در توسعه نرم‌افزار دارد ولی با رشد در توسعه و پیچیدگی، چالش‌ها و محدودیت هایی را به همراه دارد. این محدودیت‌ها فاکتورهای (اطمینان‌پذیری، مقیاس‌پذیری و نگه‌داری) در برنامه‌های فراوان_داده که در فصل قبل شرح دادیم را تحت تاثیر قرار می‌دهد و این معماری را برای توسعه‌ی همچین برنامه‌هایی ناممکن می‌سازد.

مکعب مقیاس پذیری را برای بیان روش‌های مختلف مقیاس دهی سیستم بررسی کردیم و محدودیت‌هایی که معماری یکپارچه در مقیاس دهی ایجاد می کند را مشاهده کردیم. نهایتا به صورت خلاصه فواید و ایراد های معماری میکروسرویس را ذکر کردیم. معماری میکروسرویس چالش‌ها و هزینه‌هایی برای ما ایجاد می‌کند و لحاظ کردن آن‌ها در طراحی سیستم اهمیت زیادی برای ما دارد.

فصل ۳

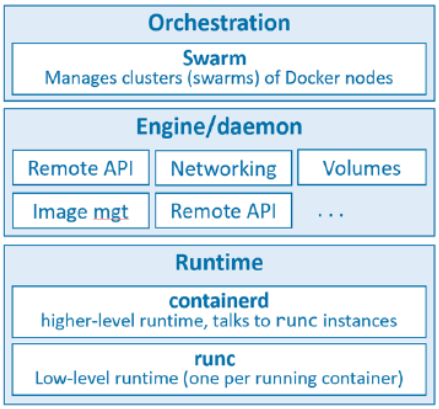
فناوری Docker

۱.۳ مقدمه

در این فصل به بررسی فناوری **Docker** می‌پردازیم .[۴] این فناوری شامل سه قسمت اصلی(شکل ۱.۳) می‌شود:

- **The runtime**
- **The daemon(a.k.a engine)**
- **The orchestrator**

که در ادامه این فصل به قسمت‌هایی از آن می‌پردازیم.



شکل ۱.۳ : معماری docker

توضیح مختصری از نصب و راه‌اندازی Docker به همراه شرح دستورات اولیه برای ایجاد **کانتینر** [۱] و **کانتینرایز کردن برنامه** [۲] می دهیم. سپس وظایف و نقش docker Engine/daemon را بررسی می‌کنیم.

در ادامه درباره‌ی **ایمیج** [۳] و کانتینر صحبت می‌کنیم و مثالی از ایجاد آن با استفاده از **Dockerfile** می‌زنیم. سپس ماشین مجازی [۴] را با کانتینر مقایسه می کنیم و تفاوت ها را ذکر می‌کنیم.

---

[۱] container
[۲] Application containerization
[۳] image
[۴] virtual machine

۲.۳ نصب و راه‌اندازی

نصب و راه‌اندازی داکر در همه‌ی سیستم عامل‌ها شامل ویندوز، لینوکس و مک به راحتی امکان پذیر است ولی باید به این نکته توجه داشت که کانتینر در حال اجرا از هسته‌ی [۵] سیستم‌عامل میزبان به صورت اشتراکی استفاده می‌کند به همین دلیل کانتینری که با اشتراک هسته ویندوز ساخته شده در سیستم عامل لینوکس قابل اجرا نیست. با این وجود در ویندوز این قابلیت در ورژن‌های جدید فراهم شده که با استفاده از **Hyper-v VM** و یا **WSL** [۶] کانتینر لینوکس را در ویندوز اجرا کنید.

برای نمونه برای نصب داکر در ubuntu linux می‌توانید از apt [۷] در ubuntu استفاده کنید:

```
$ sudo apt-get update
$ sudo apt-get install docker-ce docker-ce-cli containerd.io
```

سپس با این دستور زیر می‌توانید مطمئن بشید که داکر نصب شده است:

```
$ sudo docker --version
Docker version 19.03.8, build afacb8b7f0
$ sudo docker info
Server:
Containers: 0
Running: 0
Paused: 0
Stopped: 0
...
```

۳.۳ Docker Engine/Daemon

Docker Daemon در ابتدا به صورت یکپارچه بود ولی به مرور زمان بخش‌هایی زیادی از آن جدا شد و هر قسمت به صورت جداگانه قابلیت استفاده‌ی مجدد در فناوری‌های دیگر را پیدا کردند. وظایف اصلی docker daemon شامل: مدیریت ایمیج‌ها، ساخت ایمیج‌ها، REST API، Authentication، امنیت، فراهم کردن شبکه، Orchestration.

زمانی که یک دستور در خط فرمان داکر ارسال می‌شود این دستور توسط Docker Client تبدیل به یک API payload می‌شود و به endpoint هایی که توسط docker daemon ایجاد شده است در قالب یک POST request ارسال می‌شود. این ارتباط می‌تواند به صورت local socket یا در بستر شبکه باشد.

docker daemon پس از دریافت درخواست از طریق ارتباط CRUD-style API به صورت **gRPC** [۸] با containrd تماس می‌گیرد. سپس containerd داکر image مورد نیاز را به صورت یک OCI bundle در اختیار runc قرار می‌دهد.

در نهایت runc که مسئول ساخت container است تمام سازه‌های لازم برای ساخت (cgroups, namespacs) را از طریق رابطی که با سیستم عامل دارد فراهم می‌کند و کانتینر را می‌سازد.(شکل ۲.۳)

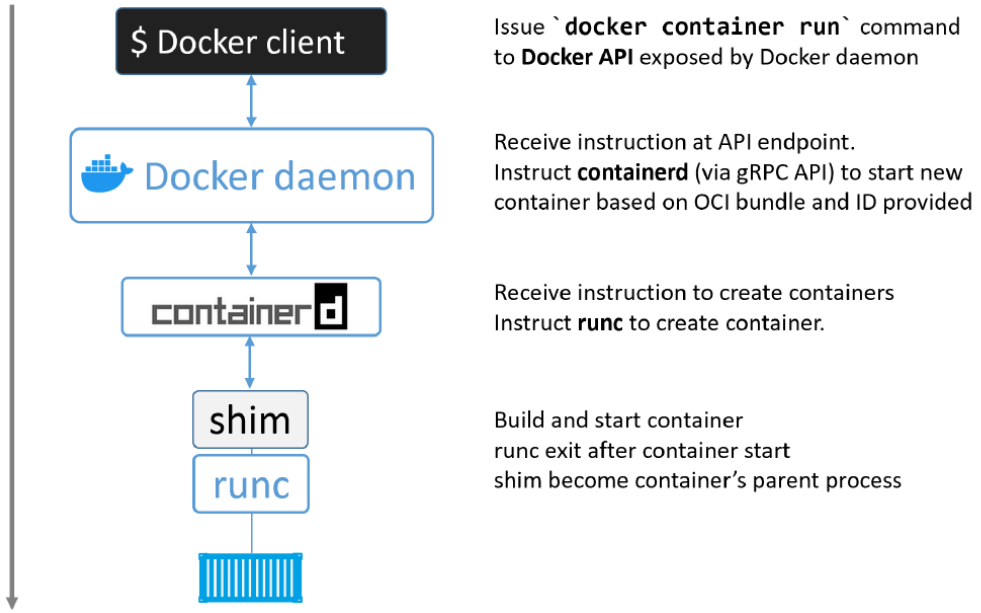
---

[۵] kernel

[۶] Windows Subsystem Linux

[۷] Package manager in debian-based linux

[۸] open-source RPC (Remote Procedure Call) framework used to build scalable and fast APIs

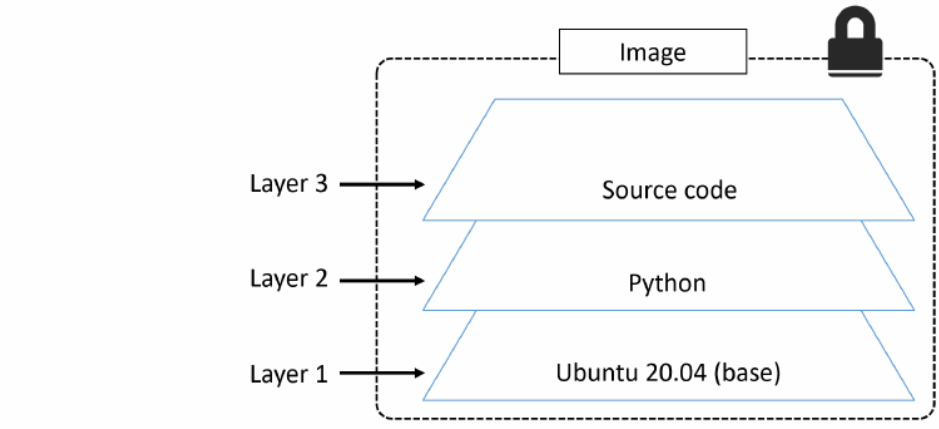


شکل ۲.۳ : پروسه‌ی ایجاد کانتینر

۴.۳ ایمیج و کانتینر

ایمیج یک بسته از تمام چیزهایی یک اپلیکیشن نیاز دارد تا اجرا شود. شامل: کد برنامه، تمام وابستگی‌ها [۹]، سازه‌های سیستم عامل. ایمیج از چندین لایه ساخته شده که رو یک دیگری قرار می‌گیرند.(شکل ۳.۳)

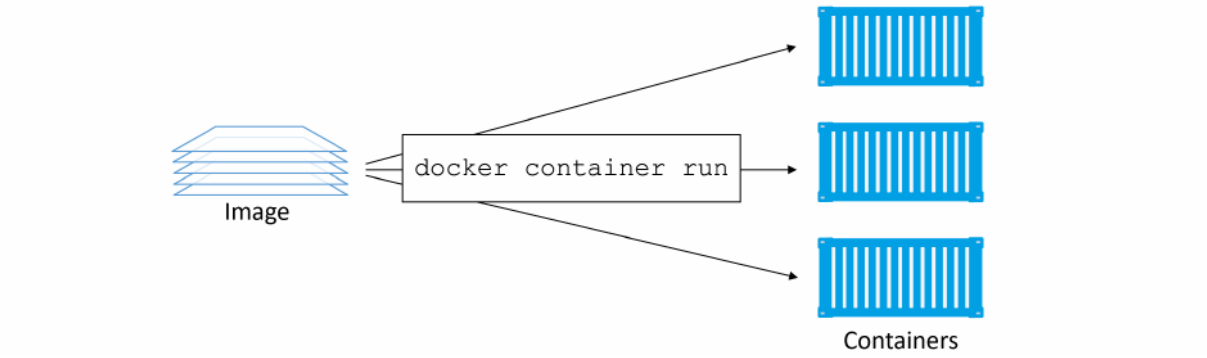
ایمیج رو می‌توان مانند کلاس‌ها در مفهوم برنامه‌نویسی در نظر گرفت و کانتینرها در واقع instance های در حال اجرا از این کلاس‌ها هستند.(شکل ۴.۳)



شکل ۳.۳: مثال از مدل لایه‌های image

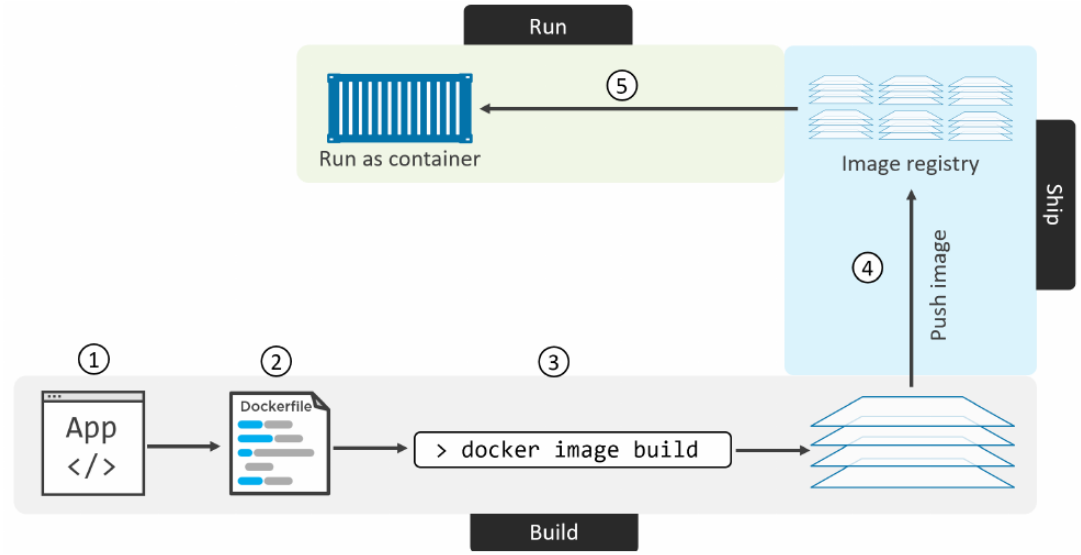
فرآیندی که معمولا برای کانتیرایز کردن یک برنامه استفاده می‌شود در شکل ۵.۳ می‌توانید ببینید. در این فرایند می‌توانیم از داکرفایل برای مشخص کردن ایمیج استفاده کنیم و تمام فایل‌های موردنیاز برای اجرای برنامه را از جمله وابستگی‌ها برای اجرا، نوع سیستم‌عامل و اضافه کردن کد برنامه. یک نمونه داکرفایل ۱.۳

[۹] dependencies



شکل ۴.۳: ایجاد چندین کانتینر از یک image

```
1.3: Sample Dockerfile
FROM alpine
LABEL maintainer="m.mehrdadshahidi@gmail.com"
RUN apk add --update nodejs nodejs-npm
COPY . /src
WORKDIR /src
RUN npm install
EXPOSE 8080
ENTRYPOINT ["node", "./app.js"]
```

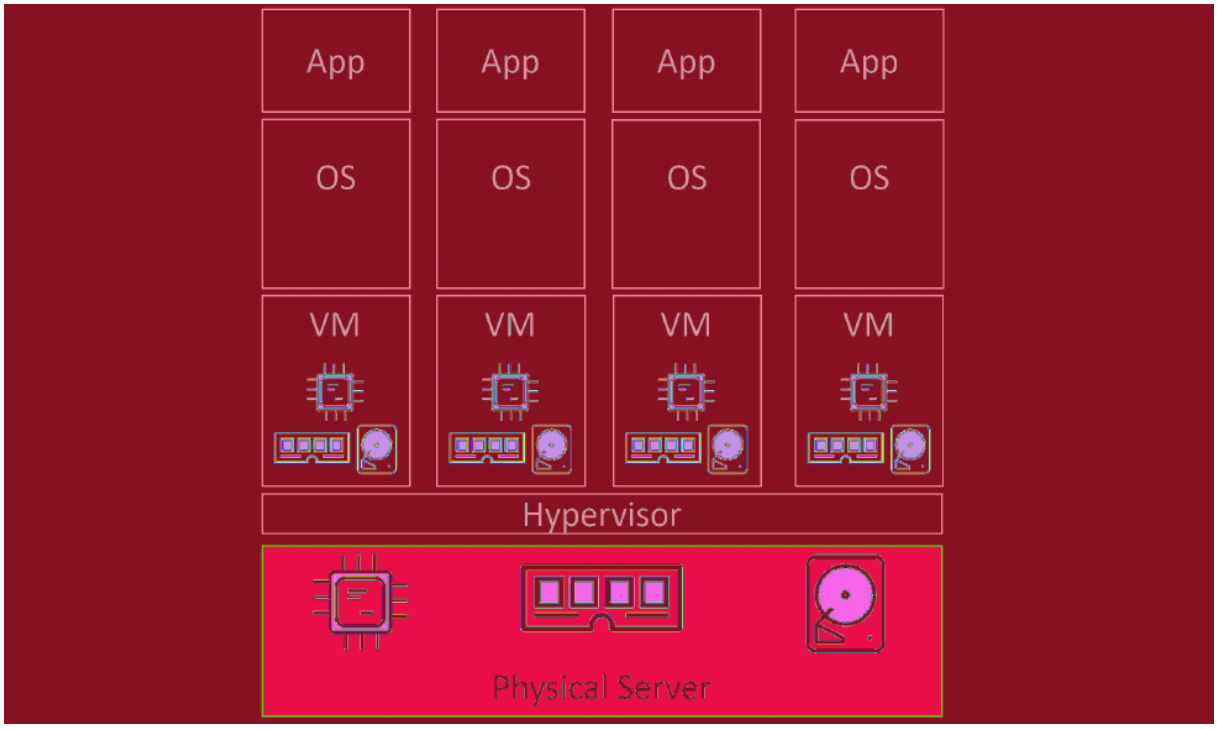


شکل ۵.۳: فرایند معمول برای کانتینرایز کردن برنامه‌ها

۵.۳ مقایسه ماشین مجازی با کانتینر

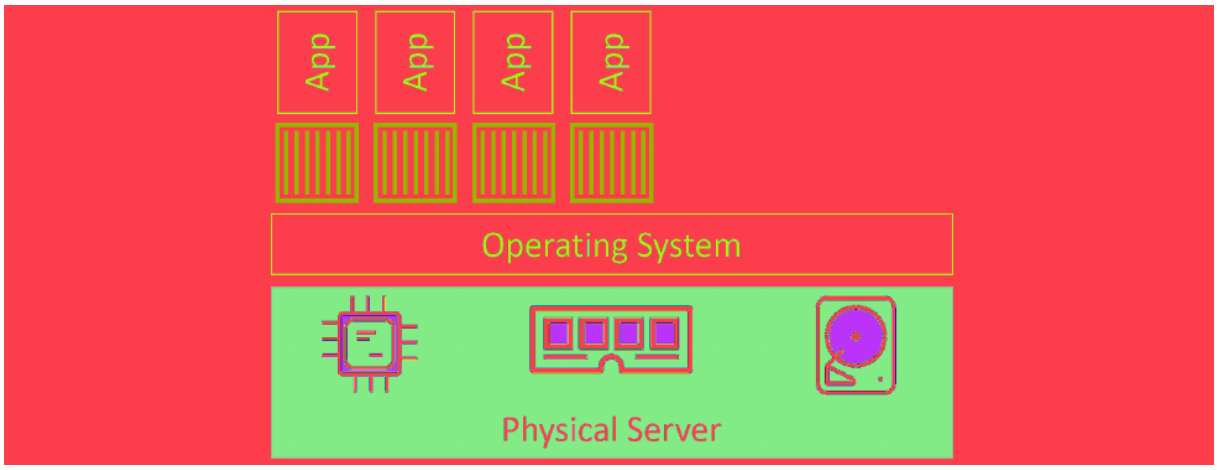
کانتینر و ماشین مجازی هردو برای اجرا به یک میزبان نیاز دارند. که این میزبان می‌تواند یک کامپیوتر PC ، یک bare metal سرور و یا یک instance روی یک فضای ابری باشد. در مدل ماشین مجازی، هر یک از این ماشین‌ها

بر روی یک hypervisor که به همه‌ی منابع سخت‌افزاری دسترسی دارد قرار می‌گیرد. (شکل ۶.۳)



شکل ۶.۳: مدل ماشین مجازی

در مدل کانتینر قضیه کمی متفاوت است. کانتینرها بر روی یک سیستم عامل میزبان قرار دارند و container engine منابع موردنیاز خودش را نظیر (Process tree, filesystems, network stacks,...) را از سیستم عامل میزبان دریافت می‌کند و در یک محیط ایزوله به اسم کانتینر قرار می‌دهد.(شکل ۷.۳)



شکل ۷.۳: مدل کانتینر

در واقع می‌توان گفت در مدل ماشین مجازی، hypervisor مجازی‌سازی در سخت‌افزار [۱۰] انجام می‌دهد ولی در مدل کانتینر مجازی سازی درسیستم عامل [۱۱] اتفاق می‌افتد.

چند مورد از فوایدی که مدل کانتینر نسبت به مدل ماشین مجازی دارد شامل:

[۱۰] hardware virtualization
[۱۱] OS virtualization

- هر ماشین مجازی برای دریافت منابع خود به یک سیستم عامل کامل [۱۲] نیاز دارد
- هر سیستم عامل در ماشین مجازی سرباری [۱۳] برای اجرا خود سیستم عامل دارد (CPU,Memory,Storage,...) (در مدل ماشین مجازی)
- برخی از سیستم عامل ها نیاز به لایسنس دارند(در مدل ماشین‌مجازی)
- نیاز به افرادی برای نگه داری و به روز رسانی سیستم عامل ها در (مدل ماشین‌مجازی)
- بزرگ ترشدن سطح حمله [۱۴] (در مدل ماشین‌مجازی)
- کوچک شدن حجم کانتینر ها (در مدل کانتینر)
- سریع تر شدن اجرای یک کانتینر (در مدل کانتینر)
- هزینه کمتر برای منابع، لایسنس، افراد (در مدل کانتینر)

۶.۳ خلاصه

در این فصل به توضیح مختصری از فناوری داکر پرداختیم. قمست‌های اصلی داکر را بررسی کوتاهی کردیم و نحوه ی نصب و راه اندازی داکر در سیستم عامل لینوکس را مشاهده کردیم.

فناوری داکر تغییرات زیادی در چند سال اخیر داشته است و سعی کرده است ماژول های مختلف خود را از یک دیگر تا حد خوبی جدا سازی کند و این کار امکان استفاده مجدد هر یک از این ماژول ها را به صورت جداگانه در فناوری‌های دیگر مانند **kubernetes** را به ما می‌دهد. علاوه بر آن، بتوانیم از طریق بستر شبکه و با استفاده از API call از ماژول‌های مختلف داکر استفاده کنیم.

فرایند ایجاد کانتینر را بررسی کردیم و وظایف هر قسمت از این ماژول‌ها در این فرایند دیدیم. نحوه‌ی ایجاد یک ایمیج با استفاده از یک داکرفایل را شرح دادیم و فرایندی معمولی که برای کانتینرایز کردن یک برنامه استفاده می‌شود را نشان دادیم.

در انتها مدل ماشین مجازی را با مدل کانتینر مقایسه کردیم و فوایدی که کانتینر ها نسبت به ماشین مجازی دارند را ذکر کردیم. ویژگی‌هایی از نظیر سایز کوچک و سرعت بالا در شروع به اجرا کردن کانتینر ها، این مدل را مدلی مناسب برای معماری های میکروسرویس می‌کند و به ما اجازه می دهد که این سرویس ها را با مقیاس بالا در یک شبکه توزیع شده پیاده سازی کنیم.

[۱۲] full-blown
[۱۳] overhead
[۱۴] attack surface

فصل ۴

فناوری GraphQL

۱.۴ مقدمه

GraphQL یک استاندارد برای API است که یک جایگزین سریع و انعطاف‌پذیر برای **REST** به حساب می‌آید.[۲] این فناوری توسط facebook توسعه یافته و به صورت متن باز [۱] توسط جمعی از سازمان‌ها و افراد در سراسر دنیا نگه‌داری می‌شود.

نقطه‌ی متمایز کننده‌ی این فناوری نسبت به REST استفاده از یک زبان تعریفی [۲] گزارش گیری [۳] است. در این فصل به توضیح مختصری از کارکرد این فناوری و مقایسه آن با فناوری پیشین خود می‌پردازیم.

۲.۴ چالش‌های REST API

امروزه اکثر برنامه‌ها این نیاز را دارند که از سرور داده‌هایی که در پایگاه داده ذخیره شده اند را دریافت کنند. API ها این وظیفه را دارند که یک رابط متناسب با نیازهای برنامه تهیه کنند. فناوری REST یک فناوری محبوب برای فراهم کردن داده ها در سمت سرور بوده است. در حین توسعه این فناوری معمولا نیازهای برنامه‌ها ساده و سرعت توسعه به مراتب کمتر از امروزه بوده است.اما حوزه‌ی API تغییرات اساسی در چند سال گذشته داشته است.

به صورت خاص، سه فاکتوری که روش طراحی API ها را به چالش انداخته اند شامل:

- نیاز به بارگیری سریع داده به دلیل افزایش استفاده از دستگاه‌های متحرک
- تنوع در قالب های و پلتفرم‌های frontend
- توسعه‌ی سریع و افزایش انتظارات در اضافه کردن امکانات جدید به نرم‌فزار

۳.۴ مقایسه‌ی فناوری GraphQL و RESTful APIs

در بخش قبل توضیح دادیم که نیاز های امروزه چه چالش هایی برای ما به وجود آورده است. در این قسمت با توضیح یک مثال به مقایسه دو فناوری REST و GraphQL می پردازیم. برای نشان دادن تفاوت این دو فناوری در قسمت دریافت داده‌ها مثال یک برنامه بلاگ را در نظر بگیرید. برنامه نیاز دارد تا عنوان پست‌های یک کاربر خاص را نمایش دهد.

---

[۱] Open-source
[۲] declarative
[۳] query

در REST API معمولا برای دریافت اطلاعات **نقاط پایانی** [۴] مختلفی تعریف می‌شود. در این مثال این نقاط پایانی به اینصورت می‌تواند باشد:

این نقطه پایانی برای دریافت اطلاعات اولیه کاربر استفاده می‌شود.

```
/users/<id>:
```

این نقطه پایانی برای دریافت تمام پست‌های کاربر استفاده می‌شود

```
/users/<id>/posts
```

این نقطه پایانی برای دریافت لیست دنبال کننده‌های یک کاربر مورد استفاده قرار می‌گیرد.

```
/users/<id>/followers
```

همانطور که در شکل ۱.۴ نشان داده شده است. در این حالت، ما باید سه درخواست به سه نقطه پایانی مختلف بفرستیم. ضمن اینکه ما اطلاعاتی بیش از نیاز خود دریافت می‌کنیم. (overfetching)

از طرفی در حالت استفاده از GraphQL ما فقط یک گزارش که شامل اطلاعات موردنیاز است به سرور میفرستیم و سرور یک پاسخ به صورت JSON که شامل تمام اطلاعات مورد نیاز است، ارسال می‌کند.(شکل ۲.۴)

۴.۴ مفاهیم پایه در GraphQL

در این بخش به توضیح مختصری از مبانی ساختارهای زبانی GraphQL می‌پردازیم. سپس چگونگی تعریف نوع‌ها و همچنین فرستادن گزارش و تغییرات (mutations) را بررسی اجمالی خواهیم کرد.

۱.۴.۴ زبان تعریف طرح (Schema Definition Language)

GraphQL سیستم نوع مخصوص خود برای تعریف کردن طرح یک api را دارا می‌باشد. نحو برای نوشتن طرح SDL نامیده می‌شود.

یک مثال ساده برای تعریف نوع Person به صورت زیر می‌باشد:

```
type Person {
    name: String!
    age: Int!
}
```

و برعکس آن، ارتباط نیاز در سمت نوع Person نیز تعریف شود:

[۴] endpoints

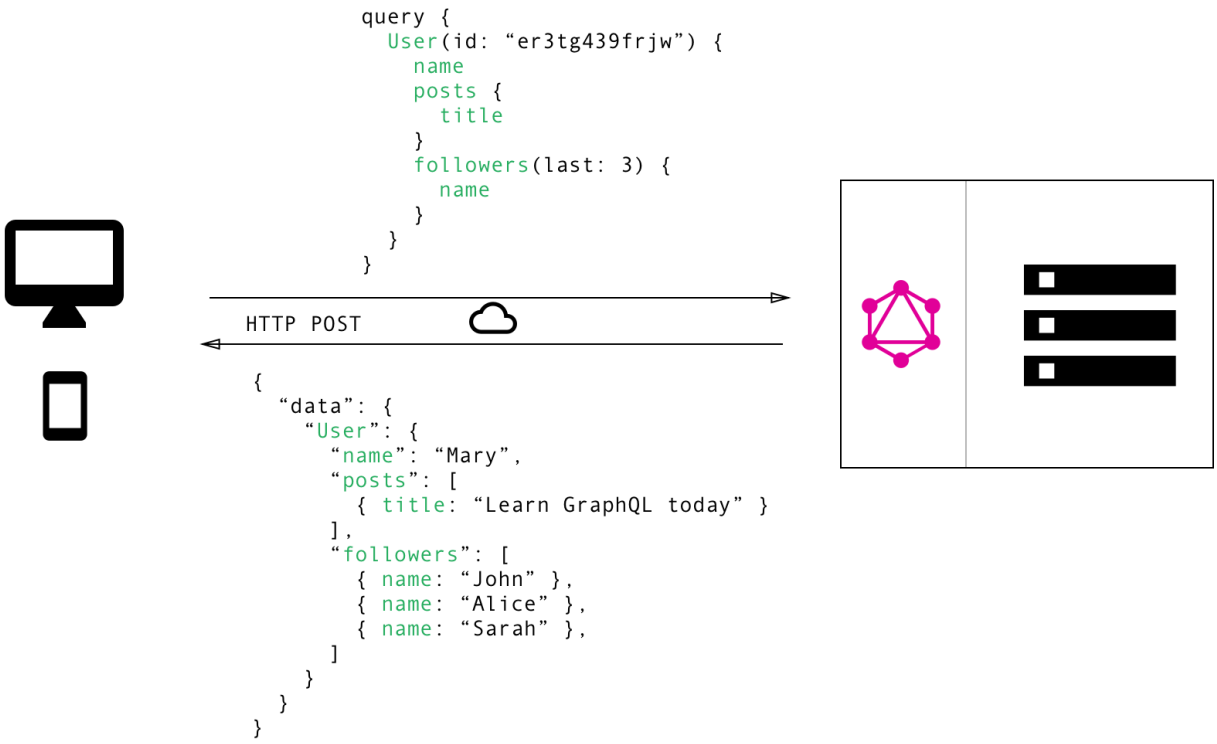
شکل ۱.۴: در REST ۳ درخواست متفاوت برای دریافت داده‌ی موردنیاز

```
type Person {
  name: String!
  age: Int!
  posts: [Post!]!
}
```

۲.۴.۴ دریافت داده با گزارش گیری

همانطور که گفته شد در GraphQL فقط یک نقطه پایانی وجود دارد. در نتیجه نیاز است که استفاده کننده با فراهم کردن اطلاعات بیشتر، نیازهای داده‌ایی خود را مشخص کند. این اطلاعات، گزارش گیری نامیده می‌شود. در زیر یک مثال ساده از گزارش گیری را مشاهده می‌کنید:

```
{
  allPersons {
    name
  }
}
```



شکل ۲.۴: در GraphQL کاربر می‌تواند به صورت دقیق مشخص کند که چه داده‌ای می‌خواهد

allpersons یک فیلد در گزارش است که فیلد ریشه‌ایی نامیده می‌شود. و هر چیزی که به دنبال آن می‌آید، payload نامیده می‌شود. تنها فیلدی که در payload ذکر شده فیلد name بوده است.

این گزارش لیستی از person ها را بر می‌گرداند. پاسخ آن به صورت زیر می‌باشد:

```
{
  ”allPersons”: [
    { ”name”: ”Johnny” },
    { ”name”: ”Sarah” },
    { ”name”: ”Alice” }
  ]
}
```

یکی از مزیت های گزارش گیری در GraphQL استفاده از گزارش‌های تو در تو است. در قسمت پروژه پیاده شده، بیشتر با این گزارش‌ها آشنا خواهیم شد.

۳.۴.۴ نوشتن داده با Mutation

نیازی که اکثر برنامه ها دارند،ایجاد تغییرات در اطلاعات، حذف داده و یا نوشتن اطلاعات جدید در سرور است. برای انجام این کار در GraphQL از mutation می‌توان بهره برد.

mutation از لحاظ نحوی کاملا شبیه گزارش گیری است، با این تفاوت که همیشه با کلمه کلیدی mutation شروع می‌شود. ساختن person جدید می‌تواند به صورت زیر باشد:

```
mutation {
    createPerson(name: "Bob", age: 36) {
        name
        age
    }
}
```

مانند گزارش گیری، یک فیلد ریشه‌ایی خواهیم داشت. که در مثال بالا createPerson است. همچنین باید ارگومنت‌های مربوطه نیر مشخص شوند. در این مثال name و age ورودی های مربوط بوده اند. سپس مانند گزارش گیری کاربر می‌تواند فیلد مورد نیاز را از سرور درخواست کند. در مثال ما، مجددا name و age داده‌های درخواستی بوده‌اند، تا کاربر بتواند از صحت تغییرات اطمینان حاصل کند.

```
"createPerson": {
  "name": "Bob",
  "age": 36,
}
```

در قسمت توضیحات پروژه، مثال‌های پیچیده تری از mutation خواهیم داشت.

۵.۴ خلاصه

در این فصل به شرح فناوری GraphQL پرداختیم. ابتدا به چالش‌هایی که در REST وجود داشت اشاره کردیم. مشکلات REST شامل:

- overfetching
- underfetching
- عدم انعطاف پذیری به تغییر در frontend

را ذکر کردیم.

در ادامه با ذکر مثالی به مقایسه این دو پرداختیم و نهایتا مفاهیم پایه GraphQL شامل: تعریف نوع، نحوه‌ی گزارش گیری و تغییر (mutation) را مختصرا شرح دادیم.

فصل ۵

پیاده‌سازی پروژه‌ی نمونه

۱.۵ مقدمه

در این فصل تلاش بر این است با استفاده از مطالب و فناوری‌هایی که در فصول قبل مورد بحث قرار گرفت یک پروژه نمونه پیاده سازی کنیم. هدف از انجام این پروژه‌ی نمونه آشنایی با چالش‌های پیاده سازی این فناوری ها و ظرفیت هایی که به ما می‌دهند تا اپلیکیشنی مقیاس‌پذیر و انعطاف‌پذیر به تغییر داشته باشیم.

برنامه‌ای که قرار است پیاده‌سازی کنیم مدل مشابه توییتر خواهد بود و از آن با نام شوییتر یاد خواهیم کرد. در این پروژه از یک سرور مجازی [۱] ابری در آلمان به عنوان ماشین میزبان استفاده کردیم و سعی شده است همه قسمت های موردنیاز برای اینکه یک وب اپلیکیشن واقعی و کامل ولی در عین حال ساده داشته باشیم پیاده سازی شود.

۲.۵ ابزار و فناوری‌ها

ابزار و فناوری‌هایی که در این پروژه استفاده کردیم شامل موارد زیر می‌شوند:

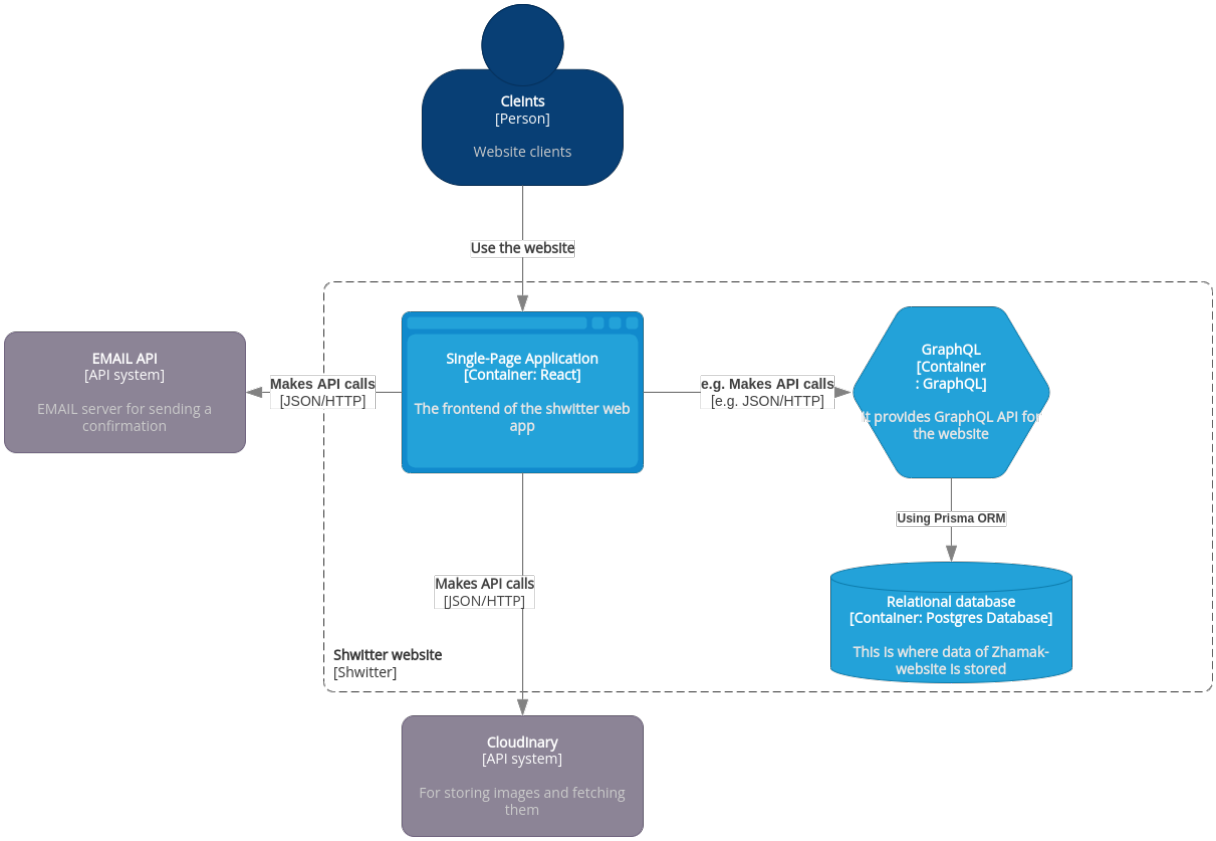
- Ubuntu linux به عنوان سیستم عامل برای سرور
- named سرویس به عنوان DNS سرور
- nginx به عنوان وب سرور و proxy reverse برای سرویس‌ها
- استفاده از certbot برای گرفتن گواهی‌نامه SSL
- PostgreSQL به عنوان پایگاه داده(در داخل docker)
- **Backend:**
  - Apollo سرور HTTP برای GraphQL
  - GraphQL Nexus
  - Prisma Client
  - Prisma Migrate
- **Frontend:**
  - React

[۱] VPS

- React-Hook-Form
- Yup
- TailwindCss

۳.۵ معماری کلی پروژه نمونه

در این پروژه از نمودار C4 برای نشون دادن معماری برنامه در سطح سیستم استفاده شده است.(شکل ۱.۵)



شکل ۱.۵ : معماری نرم‌افزار با استفاده از نمودار C4

GraphQL ۴.۵

در پیاده‌سازی این پروژه روش schema first اتخاذ می‌کنیم به این معنی که ابتدا شروع به تعریف یک GraphQL schema که رابط بین سرور و front است می‌کنیم. در این پروژه از کتابخانه Apollo که یک API برای GrapghQL است استفاده می‌کنیم تا با frontend ارتباط برقرار کنیم. schema برای برنامه به شرح زیر است:

```
type Query {
  allUsers(data: FilterInputType): AllUsers!
  feed(filter: String, orderBy: [ShweetOrderByInput!], skip: Int,
   take: Int): Feed!
  me: User
  user: User
}
type Mutation {
  deleteShweet(id: Int!): Shweet!
  editProfile(data: ProfileInput!): Profile
  like(shweetId: Int!): LikedShweet
  login(email: String!, password: String!): AuthPayload!
  shweet(content: String!): Shweet!
  signup(email: String!, name: String!, password: String!, username
   : String!): AuthPayload!
  updateShweet(content: String!, id: Int!): Shweet!
}
type Feed {
    count: Int!
    shweets: [Shweet!]!
}
type LikedShweet {
likedAt: DateTime!
shweet: Shweet
shweetId: Int!
user: User
userId: Int!
}
type Profile {
avatar: String
bio: String
createdAt: DateTime
location: String
name: String
userId: Int!
website: String
}
type Shweet {
author: User
content: String!
createdAt: DateTime!
id: Int!
likedShweet: [LikedShweet!]!
updatedAt: DateTime!
}
```

```
type User {
email: String!
id: Int!
profile: Profile
shweets: [Shweet!]!
username: String!
}
```

۵.۵ Docker

در این پروژه از فناوری Docker در چندین قسمت استفاده شده :

- backend containerization
- frontend containerization
- database containerization
- database migration(temporarily)
- Github action for deployment on server

Dockerfile برای ساخت image هر یک از سرویس‌های برنامه به شرح زیر است:

Backend service

```
FROM node:16-alpine as builder
RUN apk add --no-cache libc6-compat
WORKDIR /app
COPY package.json ./
RUN yarn install
COPY . .
RUN yarn generate
RUN yarn tsc

FROM node:16-alpine
WORKDIR /app
COPY --from=builder /app/node_modules ./node_modules
COPY --from=builder /app/package*.json ./
COPY --from=builder /app/.env ./
COPY --from=builder /app/dist ./dist
RUN chown -R node:node /app/dist
USER node
EXPOSE 4000
CMD  ["yarn", "start"]
```

Frontend service

```
FROM node:16-alpine
WORKDIR /app
COPY package.json ./
RUN yarn install --frozen-lockfile
COPY . .
COPY . /app
RUN mkdir build
EXPOSE 3000
CMD [”yarn”, ”start”]
```

برای اجرا کردن سرویس‌ها و مشخص کردن بستر شبکه برای ارتباط آن‌ها از docker compose استفاده می کنیم. docker compose شامل یک yaml فایل به شرح زیر است:

docker compose

```
services:
    back:
        container_name: shwit-graph
        build:
        context: ./back
        dockerfile: ./prod.Dockerfile
        environment:
        HOST: db
        ports:
        - ”${PORT:-4000}:4000”
        command: [”yarn”, ”start”]
        depends_on:
        - db
        networks:
        - shwitter

    migrate_db:
        container_name: shwit-migrate
        build:
        context: ./back
        dockerfile: ./Dockerfile
        environment:
        HOST: db
        command: [”yarn”, ”prisma”, ”migrate”, ”deploy”]
        depends_on:
        - db
        networks:
        - shwitter
```

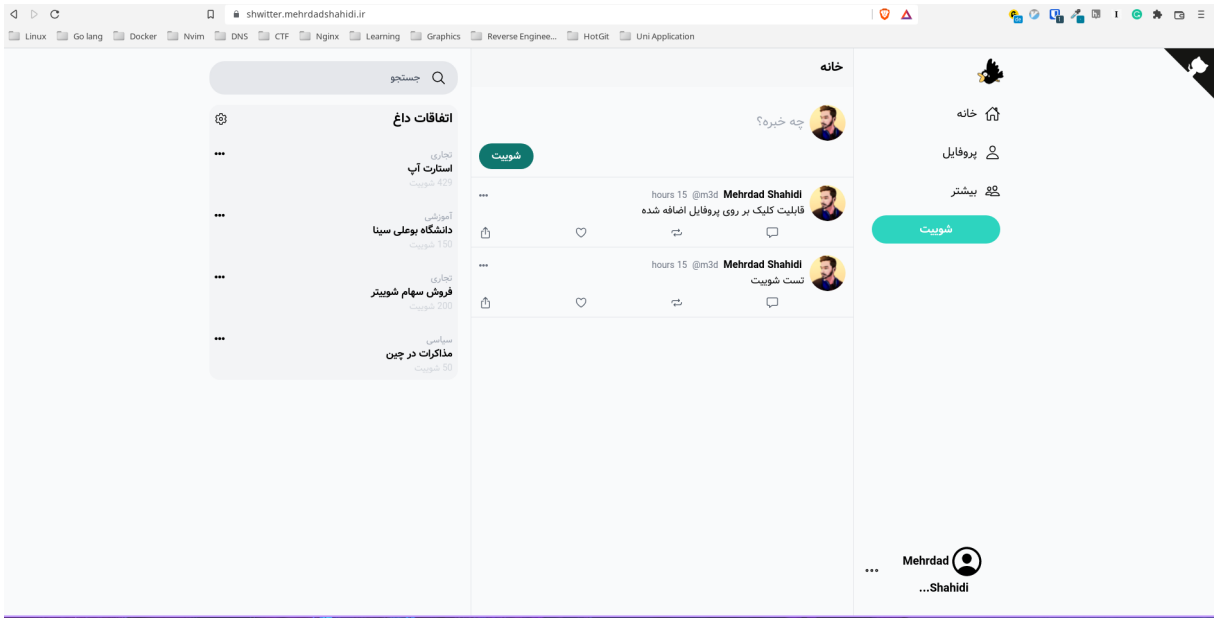
docker compose

```
  db:
        image: postgres:14.1-alpine
        container_name: shwit-db
        env_file:
        - ./back/.env
        # environment:
        #   POSTGRES_USER: ${DATABASE_USERNAME}
        #   POSTGRES_PASSWORD: ${DATABASE_PASSWORD}
        #   POSTGRES_DB: ${DATABASE_NAME}
        ports:
        - "${DB_PORT:-5432}:5432"
        volumes:
        - postgres_prod:/var/lib/postgresql
        networks:
        - shwitter

    front:
        container_name: shwit-ui
        build:
        context: ./front
        dockerfile: ./prod.Dockerfile
        command: >
        sh -c "yarn build"

        volumes:
        - /var/www/shwitter:/app/build
        ports:
        - "${FRONT_PORT:-3000}:3000"

    volumes:
    postgres_prod:

    networks:
    shwitter:
        name: 'shwitter-net'
```

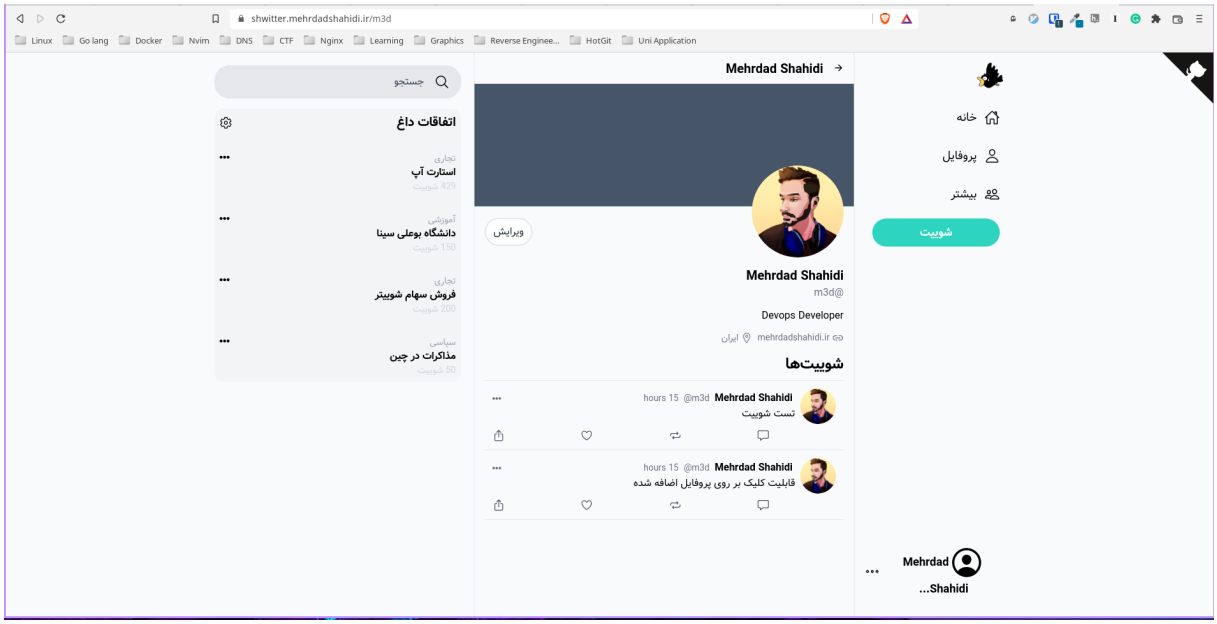
۶.۵ رابط کاربری

برای پیاده سازی frontend از React استفاده شده است و همینطور به صورت مستقیم و تعاملی می‌توانید به صورت مستقیم با GraphQL ارتباط برقرار کنید.

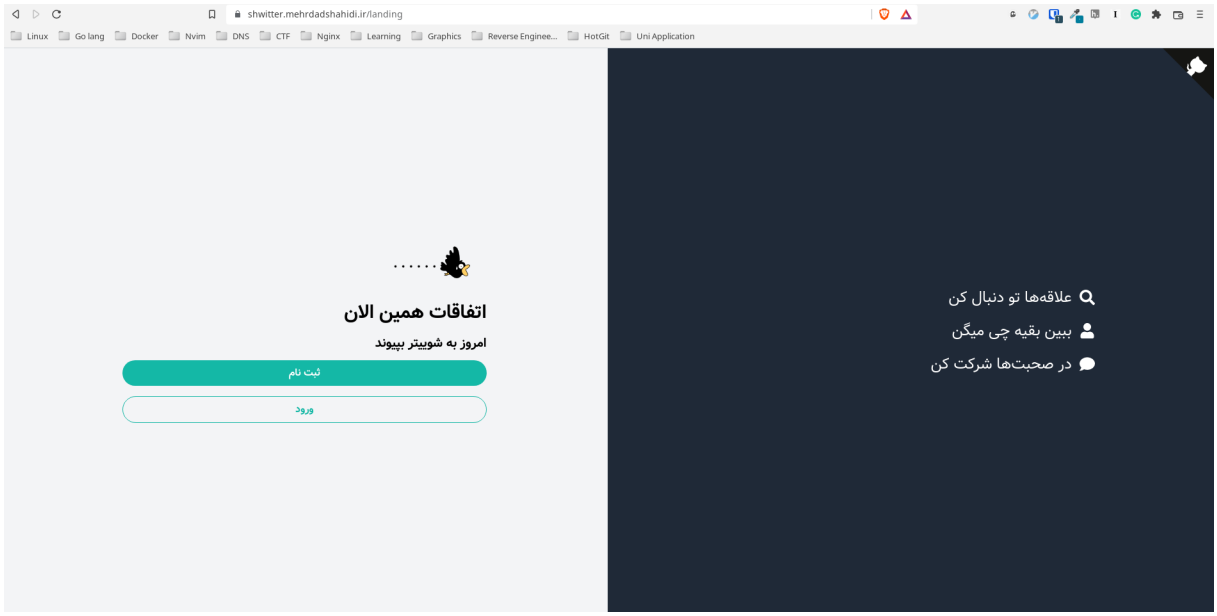
خروجی پروژه رو می‌توانید در این دامنه shwitter.mehrdadshahidi.ir مشاهده کنید.



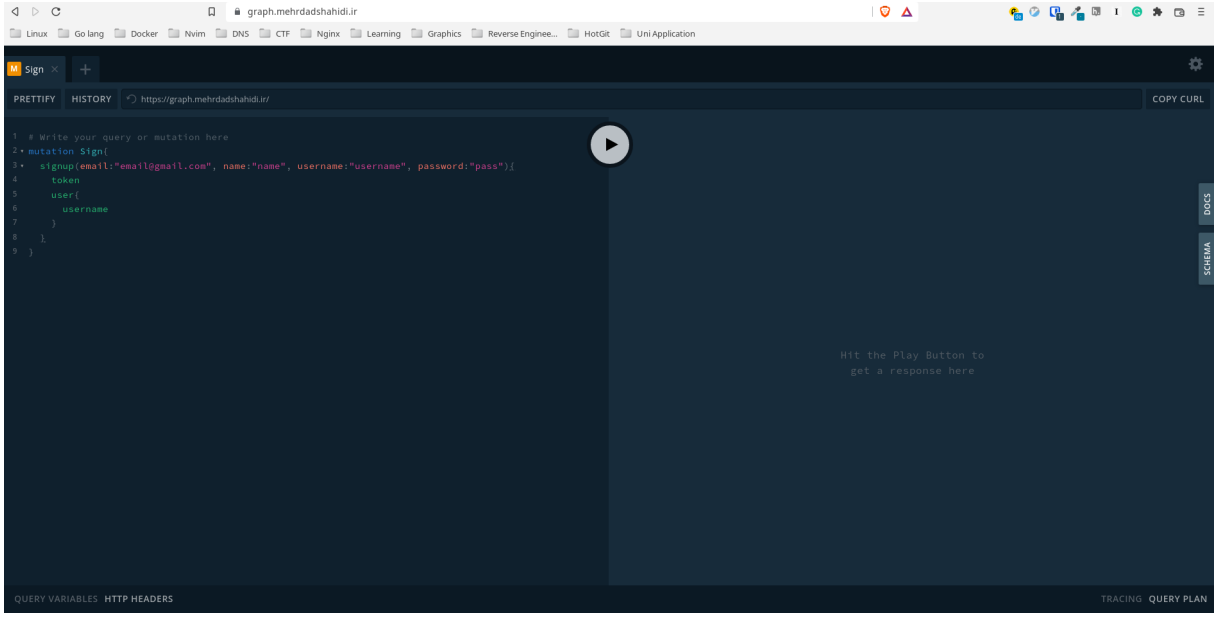
شکل ۲.۵: اسکرین شات از رابط کاربری



شکل ۳.۵: اسکرین شات از رابط کاربری



شکل ۴.۵: اسکرین شات از رابط کاربری



شکل ۵.۵: ارتباط مستقیم با GraphQL

مراجع

[1] Farley, D. *Modern software engineering: doing what works to build better software faster*. Addison-Wesley, 2021.

[2] HowToGraphQL. Graphql. https://www.howtographql.com, 2021.

[3] Kleppmann, M. *Designing Data-Intensive Applications*. O'Reilly, 2017.

[4] Poulton, N. *Docker deep dive: zero to Docker in a single book*. 2020.

[5] Richardson, C. *Microservices patterns*. Manning Publications, 2018.

Bu-Ali Sina University

Engineering Department

Computer Engineering

B.Sc Thesis

Trending Technologies in Data-intensive Application Development And Sample Implementation

MohammadMehrdad Shahidi

supervised by

Dr. Mahdi Sakhai nia

Fall 2021