



دانشگاه صنعتی امیرکبیر

(پلی تکنیک تهران)

دانشکده برق

پروژه کارشناسی

گرایش الکترونیک

طراحی و پیاده سازی بستر بدون سرور اینترنت اشیاء

نگارش

محمدحسن مجاب

استاد راهنما

دکتر طاهری

شهریور ۹۸

بِسْمِ اللَّهِ الرَّحْمَنِ الرَّحِيمِ



دانشگاه صنعتی امیرکبیر
(پلی تکنیک تهران)

به نام خدا

تعهدنامه اصالت اثر

تاریخ: شهریور ۹۸

اینجانب **محمدحسن مجاب** متعهد می‌شوم که مطالب مندرج در این پایان‌نامه حاصل کار پژوهشی اینجانب تحت نظارت و راهنمایی اساتید دانشگاه صنعتی امیرکبیر بوده و به دستاوردهای دیگران که در این پژوهش از آنها استفاده شده است مطابق مقررات و روال متعارف ارجاع و در فهرست منابع و مآخذ ذکر گردیده است. این پایان‌نامه قبلاً برای احراز هیچ مدرک هم‌سطح یا بالاتر ارائه نگردیده است. در صورت اثبات تخلف در هر زمان، مدرک تحصیلی صادر شده توسط دانشگاه از درجه اعتبار ساقط بوده و دانشگاه حق پیگیری قانونی خواهد داشت.

کلیه نتایج و حقوق حاصل از این پایان‌نامه متعلق به دانشگاه صنعتی امیرکبیر می‌باشد. هرگونه استفاده از نتایج علمی و عملی، واگذاری اطلاعات به دیگران یا چاپ و تکثیر، نسخه‌برداری، ترجمه و اقتباس از این پایان‌نامه بدون موافقت کتبی دانشگاه صنعتی امیرکبیر ممنوع است. نقل مطالب با ذکر مآخذ بلامانع است.

محمدحسن مجاب

امضا

سپاسگزاری

از جناب آقای دکتر طاهری که بنده را یاری نمودند و جناب آقای دکتر شریفیان بابت راهنمایی‌های
ارزنده ایشان کمال تشکر را دارم.

محمد حسن مجاب
شهریور ۹۸

چکیده

با توجه به گسترش اینترنت اشیاء در زمینه های تحقیقاتی و کاربردی، احساس نیاز به سرویس های ابری اینترنت اشیاء بیشتر شده است. بیشتر شرکت ها و استارت آپ هایی که در این حوزه فعال هستند نیز تنها در سطح سخت افزار و توسعه نرم افزار فعالیت می کنند و برای ذخیره سازی داده ها و پردازش ها از ارائه دهنده های ابری استفاده می کنند. همزمان با اینترنت اشیاء، معماری بدون سرور نیز در زمینه رایانش ابری در حال رشد است. در این معماری تمامی هزینه و تقاضای سرورها به عهده ی ارائه دهنده ی سرویس است و تنها مدت زمان اجرای توابع معیار قرار می گیرد. همچنین توسعه دهنده ها نیازی به در نظر گرفتن شرایط زیرساخت و سرورها ندارند. هدف این پروژه راه اندازی یک بستر بدون سرور برای اجرای توابع و پردازش های شبکه های اینترنت اشیاء شامل دریافت و تحلیل داده های حسگرها و اجرای سناریوهای مشخص است. این بستر می تواند مورد استفاده استارت آپ ها، دانشگاه ها و مراکزهای تحقیقاتی اینترنت اشیاء قرار گیرد.

واژه های کلیدی:

اینترنت اشیاء، بدون سرور، تابع به عنوان سرویس

فهرست مطالب

صفحه

عنوان

۱	معرفی	۱
۱-۱	مقدمه	۲
۲-۱	تعریف مسأله	۳
۳-۱	اهداف	۳
۴-۱	ساختار	۴
۵	مفاهیم اولیه	۵
۱-۲	اینترنت اشیاء	۶
۲-۲	مدل‌های رایانش ابری	۸
۳-۲	رایانش بدون سرور	۱۰
۱-۳-۲	Backend به عنوان سرویس (BaaS)	۱۰
۲-۳-۲	تابع به عنوان سرویس (FaaS)	۱۱
۳-۳-۲	راه حل‌ها و بسترهای ابری موجود	۱۳
۴-۲	میکروسرویس و کانتینرها	۱۷
۱-۴-۲	میکروسرویس	۱۷
۲-۴-۲	کانتینرها	۲۲
۳-۴-۲	داکر	۲۳
۴-۴-۲	کوبرنتیز	۲۵
۵-۲	OpenWhisk	۳۵
۱-۵-۲	معماری OpenWhisk	۳۵
۲-۵-۲	تابع‌ها و رویدادها	۳۶
۳-۵-۲	زبان‌های برنامه نویسی OpenWhisk	۳۶
۴-۵-۲	Action ها	۳۷
۵-۵-۲	زنجیر کردن Action ها	۳۷
۶-۵-۲	نحوه عملکرد OpenWhisk	۳۹
۴۳	طراحی و پیاده سازی	۴۳
۱-۳	دید کلی	۴۴

۴۵	طراحی ۲-۳
۴۶	طراحی میکروسرویس ۱-۲-۳
۴۶	MQTT trigger provider ۲-۲-۳
۴۷	مدیریت داده‌های سری زمانی ۳-۲-۳
۴۷	برنامه کاربردی ۴-۲-۳
۴۸	تکنولوژی‌های استفاده شده ۳-۳
۴۸	MicroK8s ۱-۳-۳
۴۸	Helm ۲-۳-۳
۴۸	EMQx ۳-۳-۳
۴۸	CouchDB ۴-۳-۳
۴۹	InfluxDB ۵-۳-۳
۵۰	پیاده سازی ۴-۳
۵۰	OpenWhisk ۱-۴-۳
۵۰	MQTT trigger provider ۲-۴-۳
۵۲	سرویس‌های کوبرنتیز ۳-۴-۳
۵۳	action ها و trigger ها ۴-۴-۳
۵۴	نحوه کار و ارزیابی ۴
۵۵	نحوه کار با بستر بدون سرور اینترنت اشیا ۱-۴
۵۶	سرویس دهنده ۱-۱-۴
۵۷	سرویس گیرنده ۲-۱-۴
۵۷	ارزیابی بستر بدون سرور اینترنت اشیا ۲-۴
۵۹	جمع بندی و نتیجه گیری و کارهای آینده ۵
۶۰	جمع بندی و نتیجه گیری ۱-۵
۶۱	کارهای آینده ۲-۵
۶۲	منابع و مراجع

فهرست شکل‌ها

صفحه

شکل

۱-۲	ساختار شبکه‌ی اینترنت اشیا	۶
۲-۲	مدل‌های رایانش ابری	۹
۳-۲	مقایسه میزان کنترل توسعه دهنده در مدل‌های رایانش ابری مختلف	۱۰
۴-۲	مثال ابتدایی از معماری یک بستر بدون سرور	۱۲
۵-۲	افزایش محبوبیت کلمه serverless در Google Trends طی سه سال اخیر	۱۴
۶-۲	معماری بستر بدون سرور OpenWhisk	۱۵
۷-۲	معماری یکپارچه	۱۸
۸-۲	مقایسه معماری میکروسرویس با معماری یکپارچه	۲۰
۹-۲	توزیع میکروسرویس‌ها روی سرورهای مختلف	۲۲
۱۰-۲	نمایش یک کانتینر	۲۲
۱۱-۲	مقایسه‌ی داکر و ماشین مجازی	۲۴
۱۲-۲	رابطه‌ی نود و خوشه در کوبرنتیز	۲۷
۱۳-۲	معماری خوشه کوبرنتیز	۲۸
۱۴-۲	معماری نود کارگر	۳۰
۱۵-۲	نحوه عملکرد replica set	۳۳
۱۶-۲	روش اعلامی ایجاد شده توسط deployment	۳۴
۱۷-۲	نحوه‌ی عملکرد رویداد محور OpenWhisk	۳۵
۱۸-۲	نمای کلی ساختار action های OpenWhisk	۳۸
۱۹-۲	ساختار OpenWhisk	۳۹
۲۰-۲	چگونگی پردازش action در OpenWhisk	۴۰
۱-۳	نمای کلی معماری پروژه	۴۵
۲-۳	سرویس‌های بستر بدون سرور اینترنت اشیا	۵۱
۳-۳	سرویس‌های بستر بدون سرور اینترنت اشیا	۵۲
۴-۳	ساخت action، feed، trigger و rule	۵۳
۵-۳	آپلود web action و دریافت آدرس آن	۵۳
۱-۴	مراحل کار با بستر بدون سرور اینترنت اشیا	۵۵

۵۶	۲-۴ تابع نوشته شده برای getTelemetryKeys action
۵۷	۳-۴ آپلود، اجرا و حذف action با استفاده از دستور wsk
۵۸	۴-۴ برنامه کاربردی تلفن همراه طراحی شده

فصل اول

معرفی

۱-۱ مقدمه

وب جهان گستر^۱ طی ۲۰ سال گذشته بیش از ۷ میلیارد کاربر داشته و سرعت رشد آن بیش از ۱۰۰۰ درصد افزایش یافته است. این تکنولوژی در سیر تکامل سایر تکنولوژی هایی که روزانه با آن ها سر و کار داریم، تحول اساسی ایجاد کرده است. از تلفن های هوشمند و ساعت های هوشمند گرفته تا هواشناسی، کشاورزی، خانه های هوشمند و ماشین های هوشمند همگی تحت تأثیر این تحول قرار گرفته اند، به صورتی که می توان گفت اینترنت، شیوه زندگی کردن انسان ها را عوض کرده است.

علاوه بر کاربران عادی، دسترسی به اینترنت برای اکثر دستگاه ها و وسایل ممکن است. با پیشرفت های اخیر در زمینه الکترونیک دیجیتال و ارتباط بی سیم می توان گفت برای تمامی اشیاء این قابلیت به وجود آمده است که با اینترنت به راحتی به یکدیگر متصل شوند، با هم ارتباط داشته باشند و توسط کاربران یا اشیاء دیگر نظارت یا حتی کنترل شوند. این مفهوم به اینترنت اشیاء^۲ معروف شده است و در دهه اخیر رشد چشمگیری داشته است. در حال حاضر تعداد دستگاه هایی که به اینترنت متصل هستند از تعداد انسان ها پیشی گرفته است و پیش بینی می شود تا ۱۰ سال آینده به ۵۰۰ میلیارد برسد. دستگاه های اینترنت اشیاء در معماری، میزان حافظه و توان مصرفی با یکدیگر تفاوت دارند. شبکه های اینترنت اشیاء نیز همانند دستگاه ها و بسته به محیط متنوع هستند. همچنین این دستگاه ها در منابع محدودیت دارند که این محدودیت می تواند شامل حافظه، منبع انرژی یا توان پردازشی باشد.

دسترسی به اینترنت همچنین باعث تغییر شگرفی در حوزه خدمات فناوری اطلاعات شده است. از جمله این تغییرات می توان به پدیدار شدن رایانش ابری^۳ و افزایش محبوبیت آن اشاره کرد. رایانش ابری مدلی از ارائه سرویس است که در آن ارائه دهندگان ابری، سرویس های پردازشی از جمله فضای ذخیره سازی، سرور رایانشی و پایگاه داده را از طریق اینترنت برای کاربرها فراهم می کنند. هزینه متناسب با استفاده کاربران از این سرویس ها حساب می شود. رایانش ابری نه تنها باعث پایین آمدن هزینه های ساخت برنامه های کاربردی شده است، بلکه زمان دسترسی کاربران به زیرساخت ها و سرویس ها را کاهش داده است. در نتیجه بسیاری از شرکت ها به جای مراکز داده خصوصی خودشان به استفاده از سرویس های ابری روی آورده اند [۱۲].

¹World Wide Web

²Internet of Things

³cloud computing

۲-۱ تعریف مسأله

با توجه به گسترش اینترنت اشیاء در زمینه‌های تحقیقاتی و کاربردی و همچنین محدودیت‌های دستگاه‌های اینترنت اشیاء، احساس نیاز به سرویس‌های ابری بیشتر شده است. رایانش ابری، منابع مورد نیاز شبکه‌های اینترنت اشیاء را فراهم می‌کند. بنابراین بیشتر شرکت‌ها و استارت‌آپ‌هایی که در این حوزه هستند تنها در سطح سخت‌افزار و توسعه نرم‌افزار فعالیت می‌کنند و برای ذخیره سازی داده‌ها و پردازش‌ها از ارائه دهنده‌های ابری استفاده می‌کنند. شرکت‌های بزرگی همچون آمازون و گوگل که در زمینه رایانش ابری پیشرو هستند، سرویس‌های متنوعی در زمینه زیرساخت به عنوان سرویس تا نرم‌افزار به عنوان سرویس ارائه می‌کنند. این شرکت‌ها نمونه‌ای از ارائه دهندگان سرویس‌های ابری هستند که کاربران زیادی در سرتاسر جهان از سرویس‌هایشان استفاده می‌کنند. متأسفانه امکان دسترسی به این سرویس‌ها توسط شرکت‌های ایرانی یا اساساً وجود ندارد یا با هزینه‌های گزاف امکان پذیر است. در داخل کشور نیز روند ارائه چنین سرویس‌هایی در حال آغاز است. بنابراین در بازار داخل عرضه بسیار کمتر از تقاضا است، بنابراین نیاز به ارائه سرویس‌های نوین ابری به شدت احساس می‌شود.

استفاده از سرویس‌های ابری برای تأمین منابع دستگاه‌های اینترنت اشیاء در سطح‌های مختلف (زیرساخت، بستر نرم‌افزاری و نرم‌افزار) امکان‌پذیر است که هر کدام مزیت‌ها و معایب خاص خود را دارند. باید با بررسی دقیق‌تر و نیازسنجی، سرویس مناسب برای این حوزه مشخص گردد.

۳-۱ اهداف

همزمان با اینترنت اشیاء، معماری بدون سرور^۴ نیز در زمینه رایانش ابری در حال رشد است. در این معماری هزینه و تقاضای سرورها به عهده‌ی ارائه دهنده سرویس است و تنها مدت زمان اجرای توابع تعریف شده توسط کاربر معیار قرار می‌گیرد. همچنین توسعه دهنده‌ها نیازی به در نظر گرفتن شرایط زیرساخت و سرورها ندارند و بدون نیاز به داشتن هیچ گونه اطلاعات فنی در این حوزه به توسعه‌ی برنامه کاربردی خود با استفاده از تابع‌ها می‌پردازند.

هدف این پروژه طراحی، راه اندازی و بررسی مزایا و معایب یک بستر بدون سرور برای اجرای توابع و پردازش‌های شبکه‌های اینترنت اشیاء شامل دریافت و تحلیل داده‌های حسگرها و اجرای سناریوهای مشخص است. این بستر می‌تواند مورد استفاده استارت‌آپ‌ها، دانشگاه‌ها و مراکزهای تحقیقاتی اینترنت اشیاء قرار گیرد.

⁴serverless

۴-۱ ساختار

این پایان‌نامه به جز فصل معرفی از چهار فصل دیگر تشکیل شده است. فصل دوم به تعریف مفاهیم اولیه مورد نیاز و بررسی تکنولوژی‌ها و راه حل‌های موجود پرداخته است. پس از یادگیری مفاهیم و در نظر گرفتن اهداف پروژه، در فصل سوم، ابتدا ساختار کلی و طراحی پروژه توضیح داده شده، سپس نحوه پیاده سازی آن گام به گام مورد بررسی قرار گرفته است. در فصل چهارم نحوه‌ی کار با پروژه با ایجاد یک برنامه‌ی کاربردی تست بررسی شده و نتایج ارزیابی‌های انجام گرفته ارائه می‌شود. نهایتاً در فصل پنجم، کارهای انجام شده در پروژه جمع‌بندی و نتیجه گیری می‌شود، سپس کارهای احتمالی آینده توضیح داده خواهد شد.

فصل دوم

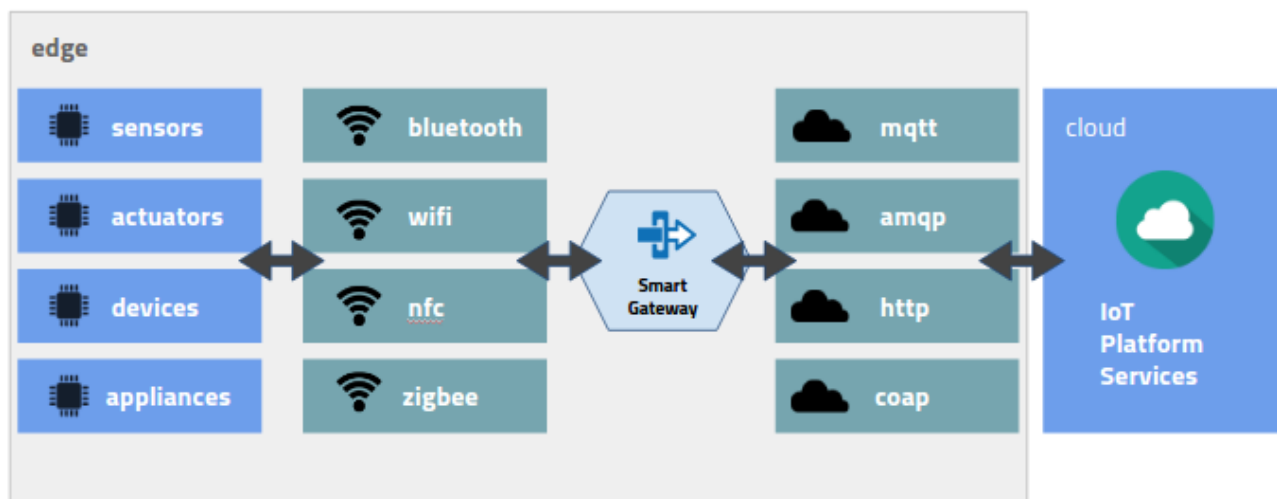
مفاهیم اولیه

در این بخش به توضیح بیشتر مفاهیم معرفی شده (اینترنت اشیاء، رایانش ابری و بدون سرور) پرداخته شده، سپس پروژه‌ها و راه حل های موجود برای توسعه هر کدام از این محیط‌ها بررسی می‌شود.

۱-۲ اینترنت اشیاء

اینترنت اشیاء شبکه‌ای از دستگاه‌های هوشمند فیزیکی مانند لوازم خانگی، وسایل نقلیه، دستگاه های پوشیدنی و حسگرها است. این دستگاه‌ها Things نامیده می‌شوند و از اینترنت برای برقراری ارتباط با یکدیگر استفاده می‌کنند. دستگاه‌های اینترنت اشیاء از نظر ماهیتی هوشمند، به هم پیوسته و محدود به منابع هستند. خانه هوشمند، کشاورزی، مراقبت‌های پزشکی و سیستم حمل و نقل هوشمند برخی از زمینه‌هایی هستند که در آینده نه چندان دور به دستگاه‌های اینترنت اشیاء متکی خواهند بود. به عنوان مثال، خودروی هوشمند از طریق ارتباط با خودروهای دیگر اطلاعات ترافیک را تجزیه و تحلیل می‌کند. در مثال دیگر پزشکان از طریق دستگاه‌های اینترنت اشیاء بر سلامتی بیماران نظارت می‌کنند. ارتباطات ذکر شده در بالا نیاز به اتصال بین دستگاه‌ها دارد و منجر به تولید حجم زیادی از داده‌ها می‌شود. با رشد اینترنت اشیاء، دنیایی در حال ایجاد است که میلیاردها دستگاه همه‌ی داده‌ها را جمع می‌کنند و از این داده‌ها می‌توان برای دستیابی به چیزی استفاده کرد که اگر همه این دستگاه‌ها به یکدیگر و اینترنت متصل نبودند، هیچگاه امکان پذیر نبود.

شکل ۱-۲ ارتباط بین انواع دستگاه‌های اینترنت اشیاء و gateway، پروتکل‌های ارتباطی و ارتباط با بستر خدمات ابری را نشان می‌دهد.



شکل ۱-۲: ساختار شبکه‌ی اینترنت اشیاء

دستگاه‌های اینترنت اشیاء ویژگی‌های بسیاری دارند که آن‌ها را از دستگاه‌های رایانشی قدیمی متمایز می‌کند. در ادامه این خصوصیات بررسی می‌شوند.

- **تنوع زیاد:** دستگاه‌های اینترنت اشیاء در سخت‌افزار و قابلیت‌های مختلف متفاوت هستند. در نتیجه، یک شبکه اینترنت اشیاء شامل دستگاه‌های مختلف برای تعامل با یکدیگر است.
- **محدودیت منابع:** دستگاه‌های اینترنت اشیاء قدرت پردازش و حافظه محدود دارند. آن‌ها برای انجام محاسبات محدود طراحی شده‌اند.
- **محدودیت انرژی:** دستگاه‌های اینترنت اشیاء می‌توانند دارای منبع انرژی مداوم یا محدود باشند. در صورت وجود انرژی مداوم، یک دستگاه به طور مداوم به منبع انرژی وصل می‌شود. از طرف دیگر، در محیط سیار، دستگاه‌های اینترنت اشیاء متحرک هستند و از باتری به عنوان منبع انرژی خود استفاده می‌کنند.
- **شبکه پویا:** اکثر دستگاه‌های اینترنت اشیاء سیار هستند که باعث می‌شود شبکه‌ی اینترنت اشیاء پویا و مرتباً در حال تغییر باشد.

۲-۲ مدل‌های رایانش ابری

رایانش ابری یک الگو است که در آن خدمات رایانشی مانند حافظه ذخیره سازی، سرورهای رایانشی و پایگاه داده ها از طریق اینترنت توسط ارائه‌دهندگان ابری^۱ به کاربران ارائه می‌شود. ارائه دهندگان برای استفاده از خدمات، هزینه را به ازای هر بار استفاده، محاسبه می‌کنند. این مدل صورتحساب شبیه به خدمات روزانه مانند برق، گاز و آب است. محیط رایانش ابری از یک ارائه دهنده ابر تشکیل شده است که منابع بر اساس تقاضا و بسیار مقیاس پذیر را از طریق اینترنت برای مشتریان فراهم می‌کند. این منابع می‌توانند یک برنامه، یک پایگاه داده یا یک ماشین مجازی^۲ ساده و بدون سیستم عامل باشند. ارائه دهنده مسئولیت بیشتر مدیریت زیرساخت ها را بر عهده دارد. با این حال، کاربر هنوز وظیفه محدود کارهایی مانند انتخاب سیستم عامل، ظرفیت و اجرای برنامه را بر عهده دارد. ارائه دهنده می‌تواند ضمن دستیابی به چند اجازه، سریعاً منابع را مستقر و مقیاس کند. در حالی که، هزینه و زحمت های مدیریتی سمت کاربر به شدت کاهش می‌یابد.

تحولات رایانش ابری را می‌توان به مدل‌های زیر تقسیم کرد:

- **زیرساخت به عنوان سرویس^۳ (IaaS)** یک مدل رایانش ابری است، که در آن ارائه دهندگان، زیرساخت را مانند سرورها، پایگاه داده و فضای مرکز داده^۴ ارائه می‌دهند. با استفاده از این مدل، کاربر نیازی به نگرانی در مورد راه اندازی، تعمیر، نگهداری و مقیاس پذیری زیرساخت‌ها ندارد.
- **بستر نرم‌افزاری به عنوان سرویس^۵ (PaaS)** متشکل از ارائه دهنده خدماتی است که بستر رایانشی آماده به همراه راه حل های کاربردی را به کاربر ارائه می‌دهد و باعث صرفه جویی در زمان راه اندازی می‌شود. مشتری می‌تواند با تعامل کمتر با واسطه برنامه های کاربردی را توسعه دهد و به سرعت پیاده‌سازی کند. مشتری همچنین نیازی به نگرانی در مورد نرم افزار، پیکربندی شبکه (provisioning) و میزبانی (hosting) ندارد.
- **نرم‌افزار به عنوان سرویس^۶ (SaaS)** یک مدل رایانش ابری است که در آن کاربران بدون هیچ گونه نگرانی در مورد پیاده‌سازی و مدیریت از برنامه های مبتنی بر وب مستقر در سرورهای ابری بهره مند می‌شوند. با استفاده از یک مرورگر وب به این برنامه های میزبان ابری دسترسی پیدا

¹cloud providers

²virtual machine

³Infrastructure as a Service

⁴data center

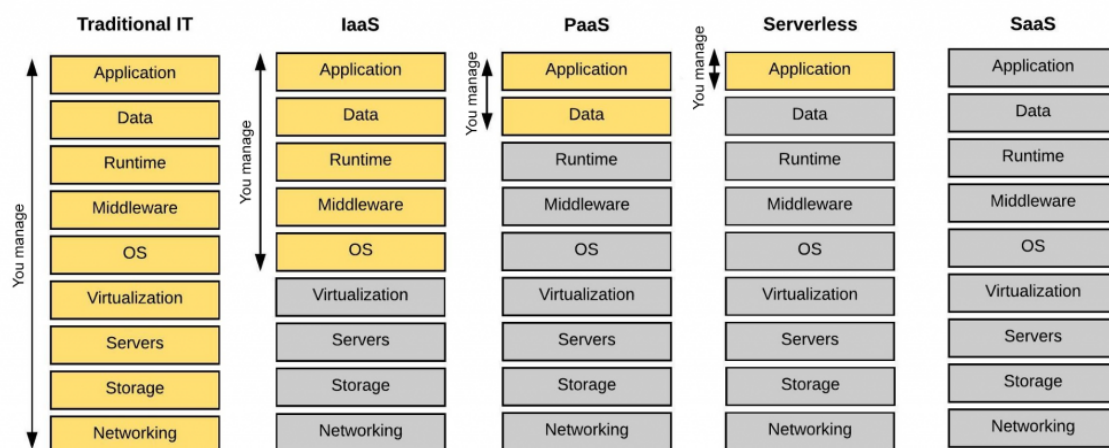
⁵Platform as a Service

⁶Software as a Service

می‌کنند. در این مدل کاربران از خدماتی با شروع سریع، تقاضا محور، موقعیت مکانی مستقل و مقیاس پذیری پویا بهره‌مند می‌شوند. با این حال، این رویکرد این مشکل را دارد که کاربر کنترل بسیار کمتری روی آن دارد.

• **رایانش بدون سرور** آخرین مدل رایانش ابری است که به طور خاص برای برنامه های گذرا (ephemeral)، فاقد وضعیت (stateless) و رویداد محور (event based) ساخته شده است. مدل رایانش بدون سرور مبتنی بر مقیاس پذیری افقی براساس تقاضا است زیرا برنامه های میزبانی شده نیاز به مقیاس شدن با سرعت بالا دارند. این مدل همچنین از رویکرد "pay as you go" رایانش ابری استفاده می‌کند و صورتحساب کاربران بر اساس استفاده با دقت میلی ثانیه حساب می‌شود. تعریف رسمی‌تر از رایانش بدون سرور این است: معماری های بدون سرور به برنامه هایی اطلاق می‌شوند که به خدمات شخص ثالث بستگی دارد (Backend به عنوان سرویس یا BaaS) یا به کدی که در کانتینرهای گذرا اجرا می‌شود (تابع به عنوان سرویس یا FaaS).

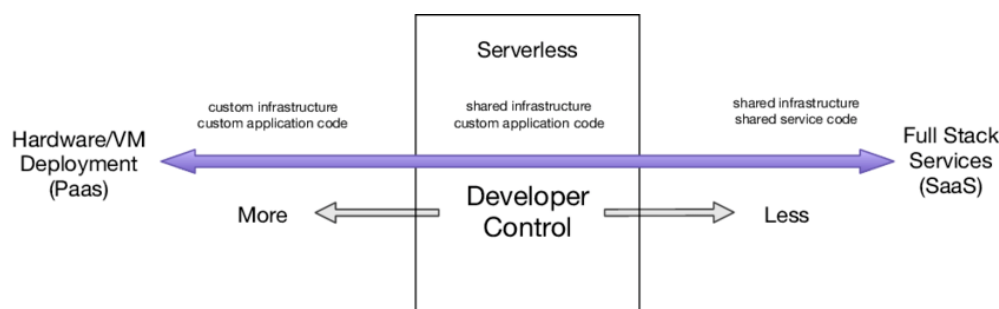
در شکل ۲-۴ مدل های رایانش ابری نشان داده شده و همچنین بخش های مدیریت شده توسط ارائه دهنده مشخص شده است.



شکل ۲-۲: مدل های رایانش ابری

۳-۲ رایانش بدون سرور

تعریف اصطلاح بدون سرور دشوار است، زیرا این اصطلاح گمراه کننده است و تعریف آن هم بر مفاهیم دیگر مانند بستر نرم افزاری به عنوان سرویس (PaaS) و نرم افزار به عنوان سرویس (SaaS) همپوشانی دارد. بدون سرور بین این دو مفهوم قرار دارد، جایی که توسعه دهنده کنترل برخی زیرساخت های ابر را از دست می دهد، اما همانطور که در شکل ۳-۲ توضیح داده شده است، بر روی کد برنامه کنترل دارد. مفاهیم بدون سرور به دو دسته تقسیم می شوند:



شکل ۳-۲: مقایسه میزان کنترل توسعه دهنده در مدل های رایانش ابری مختلف

۱-۳-۲ Backend به عنوان سرویس (BaaS)

در مدل BaaS^۷ منطق سمت سرور توسط کاربر پیاده سازی و مدیریت نشده و در عوض از سرویس های ارائه دهنده استفاده می شود. این مدل بسیار نزدیک به مفهوم نرم افزار به عنوان سرویس (SaaS) است تا سر و کار داشتن با ماشین های مجازی و کانتینرها. در این مدل، برنامه ها به اجزای کوچک تر تقسیم می شوند و تعدادی از این اجزاء کاملاً با محصولات برون سپاری شده پیاده سازی می شوند.

سرویس های BaaS دارای دامنه های عمومی هستند و از راه دور و از طریق API^۸ مورد استفاده قرار می گیرند. این سرویس ها بیشتر مورد توجه توسعه دهندگان برنامه تلفن همراه و صفحات وب ثابت قرار گرفته است زیرا به راحتی می توانند برای انجام کارهای مورد نیازشان از این سرویس ها استفاده کنند. برای مثال Firebase گوگل یک پایگاه داده است که کاملاً توسط خود گوگل مدیریت می شود و از آن مستقیماً می توان در هر برنامه کاربردی و بدون نیاز به هیچ سروری استفاده کرد.

نمونه دیگر سرویس های BaaS استفاده از منطق برنامه ای است که توسط تیم های دیگر پیاده سازی شده. به عنوان مثال سرویس هایی مانند Auth0 و Cognito برای احراز هویت کاربران و مدیریت آن ها

^۷Backend as a Service

^۸Application Programming Interface

وجود دارند که برنامه های وب و تلفن همراه می توانند از آن استفاده کنند بدون این که نیاز داشته باشند تیم توسعه آن ها حتی قسمتی از این منطق را خودشان پیاده سازی کنند.

BaaS به خاطر رشد در زمینه توسعه برنامه های کاربردی تلفن همراه معروف شد و گاهی اوقات به MBaaS⁹ شناخته می شد. اما این مفهوم به backend برای برنامه های کاربردی تحت وب و موبایل محدود نمی شود و به عنوان مثال می توان از سرویس هایی برای مدیریت سیستم فایل و ذخیره سازی داده و حتی آنالیز گفتار استفاده کرد که کاملاً توسط شرکت دیگری ارائه و مدیریت می شوند. همچنین از این سرویس ها در سمت سرور نیز می توان بهره برد.

۲-۳-۲ تابع به عنوان سرویس (FaaS)

در مدل سرویس دهی FaaS¹⁰، منطق سمت سرور همچنان توسط خود برنامه نویسان نوشته شده و کنترل می شود. اما برنامه ها در کانتینرهای فاقد وضعیت و گذرایی اجرا می شوند و که به وسیله رویدادها ایجاد شده اند.

تابع به عنوان سرویس در بسیاری از موارد برای تعریف کردن مفهوم بدون سرور استفاده می شود و بسیاری از افراد به اشتباه این دو مفهوم را به جای یکدیگر به کار می برند. FaaS به نسبت بقیه مفاهیم بدون سرور جدیدتر است و برای پیاده سازی پروژه تمرکز ما بر روی این مدل سرویس بیشتر خواهد بود، بنابراین بیشتر به توضیح کارکرد آن خواهیم پرداخت. هنگامی که می خواستیم برنامه سمت سرور را به روش های متداول قدیمی پیاده سازی کنیم، ابتدا با یک سرور میزبان، یک ماشین مجازی یا حتی یک کانتینر شروع به کار می کردیم. سپس برنامه خود را درون آن میزبان مستقر می کردیم. چنانچه میزبان یک ماشین مجازی یا کانتینر بود، برنامه ما به عنوان یک پردازش سیستم عامل اجرا می شد. معمولاً برنامه ما متشکل بود از کدهایی که برای انجام عملیات های مختلف و مرتبط با هم نوشته شده بودند.

FaaS این نوع پیاده سازی را تغییر داد. به این صورت که میزبان و پردازش برنامه از این مدل حذف شدند و تمرکز بر پیاده سازی منطق برنامه به صورت عملیات ها و تابع های جداگانه قرار گرفت. این تابع ها پس از توسعه به صورت جداگانه بر روی بستر FaaS قرار گرفته و اجرا می شوند. اما این تابع ها مانند یک پردازش سرور نیستند که پیوسته فعال باشند و در وضعیت idle قرار داشته باشند تا اینکه زمان اجرای آن ها برسد. در عوض بستر FaaS طوری ساخته شده که برای هر تابع منتظر رویداد مربوط به آن بماند. زمانی که آن رویداد رخ داد، بستر از تابع مربوط به آن یک نمونه می سازد، سپس آن نمونه را با پارامترهای آن رویداد اجرا می کند. هنگامی که اجرای تابع به اتمام رسید، بستر FaaS آزاد است که آن

⁹Mobile Backend as a Service

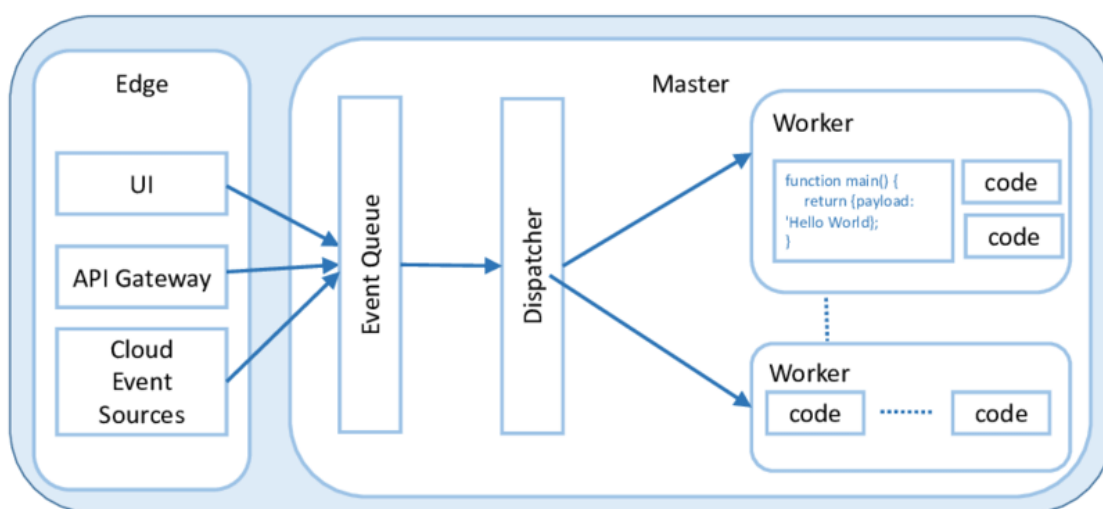
¹⁰Function as a Service

را حذف کند، اما ممکن است آن را برای بهینه سازی مدتی آن را نگه دارد تا اگر رویداد جدیدی رخ داد نخواهد آن را از ابتدا بسازد.

FaaS یک رویکرد رویداد محور^{۱۱} است که علاوه بر ارائه یک بستر برای میزبانی و اجرای توابع با بسیاری از منابع رویدادی همگام و ناهمگام ادغام شده است. برای مثال HTTP API Gateway یک منبع رویداد همگام است و صف پیام، عملیات ذخیره سازی یا رویدادهای برنامه ریزی شده، از منابع رویداد ناهمگام هستند.

از نظر سطحی، BaaS و FaaS کاملاً متفاوت هستند. مورد اول برون سپاری کامل اجزای برنامه به صورت جداگانه است و مورد دوم یک محیط میزبانی جدید برای اجرای کد شخصی افراد است. پس چرا آن ها با هم زیر مجموعه مفهوم بدون سرور قرار می گیرند؟

دلیل اصلی این است که در هر دو مورد توسعه دهنده نیاز نیست که مدیریت زیرساخت های سرور خودش و یا پردازش های سرور را در نظر بگیرد. بلکه تمامی منطق برنامه در یک محیط عملیاتی کاملاً منعطف اجرا می شود و وضعیت برنامه نیز در محیطی متشابه ذخیره می گردد. با تغییر تقاضا روی سرور، بستر بدون سرور بر این اساس تعداد ظرفیت سرورها را افزایش داده و یا کاهش می دهد، بدون اینکه لازم باشد برنامه نویسی توسط توسعه دهنده انجام شود. هزینه میزبانی یک سرویس بدون سرور معمولاً متناسب با تعداد درخواست های اجرای تابع ها محاسبه می شود. به عبارت دیگر بدون سرور به این معنی نیست که واقعا سروری در کار نباشد بلکه بدین معنی است که دیگر نیازی نیست نگران سرور باشید [۱۶].



شکل ۲-۴: مثال ابتدایی از معماری یک بستر بدون سرور

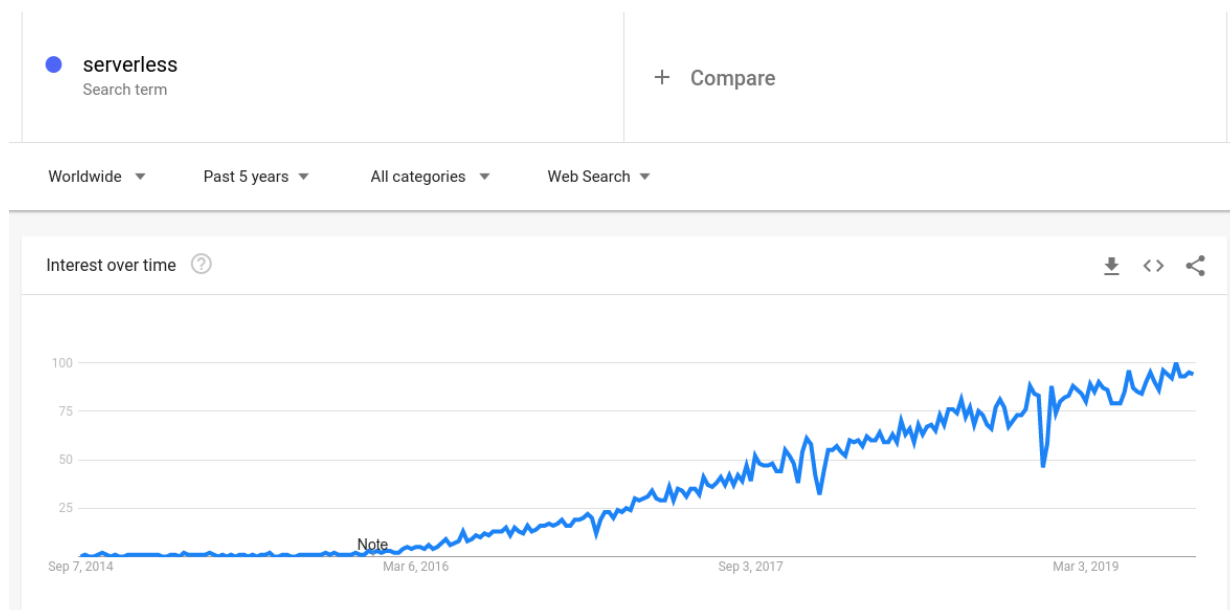
^{۱۱}event-driven

مانند آنچه در شکل ۲-۴ مشاهده می‌شود، عملکرد اصلی بستر بدون سرور متکی است به توانایی پردازش رویدادها بر اساس تقاضا، با استفاده از مجموعه ای از توابع. بستر بدون سرور باید به مدیریت مجموعه‌ای از توابع، دریافت درخواست های HTTP یا نوع دیگری از درخواست‌ها بپردازد، درخواست را با یکی از توابع مرتبط سازد، نمونه‌ای از تابع را ایجاد کند یا یک مورد موجود را پیدا کند و رویداد را به آن بفرستد، منتظر پاسخ از نمونه‌ی تابع باشد، با پاسخ از نمونه‌ی تابع به درخواست HTTP پاسخ دهد و در صورت عدم نیاز نمونه‌ی تابع را خاموش کند. گزارش‌های مربوط به کل فرآیند را نیز باید هنگام انجام تمام این کارها جمع آوری کند. چالش اصلی بستر بدون سرور ارائه همه این موارد است در حالی که باید با الزاماتی از قبیل هزینه، تحمل خطا و مقیاس پذیری کنترل شود. چالش‌هایی که یک بستر بدون سرور باید بر آن غلبه کند، عبارتند از [۱۵] [۱۱]:

- سرعت برای شروع اجرای یک تابع و پردازش رویداد.
 - صف رویدادها و سازگاری متناسب با آن‌ها. با توجه به ورود رویدادها و وضعیت فعلی صف، باید برنامه زمانبندی اجرای تابع تنظیم و مدیریت شود تا منابع را از توابعی که در حال اجرا نیستند پس بگیرد.
 - چگونگی مقیاس پذیری و مدیریت خطاها را با دقت در نظر بگیرد.
- همانطور که در شکل مشاهده می‌شود، از زمان شکل‌گیری مفهوم بدون سرور، محبوبیت آن بسیار افزایش یافته است و انتظار می‌رود که این رشد همراه با رشد IoT ادامه داشته باشد. رشد محبوبیت فناوری بدون سرور در شکل ۲-۵ قابل مشاهده است.

۳-۳-۲ راه حل‌ها و بسترهای ابری موجود

- **Amazon lambda** اولین بستری است که مدل سرویس بدون سرور را پیاده‌سازی کرد و معیارهای آن از جمله قیمت گذاری، مدل برنامه نویسی، نحوه استقرار، محدودیت منابع، امنیت و مانیتورینگ را به بازار معرفی کرد. Lambda از زبان‌های برنامه نویسی Node.js، Python، Java و C# پشتیبانی می‌کند. با توجه به این که این بستر در این زمینه پیشگام بوده است به عنوان اولین و بهترین راه حل انتخاب می‌شود و تعداد زیادی از شرکت‌های بزرگ از سرویس‌های این بستر استفاده می‌کنند.
- **Google Cloud Functions** در حال حاضر این بستر بسیار محدود است و تنها از زبان JavaScript که در محیط استاندارد Node.js نوشته شده باشد پشتیبانی می‌کند و تنها به رویدادهای سرویس



شکل ۲-۵: افزایش محبوبیت کلمه serverless در Google Trends طی سه سال اخیر

Google Cloud یا درخواست های HTTP پاسخ می‌دهد. همچنین این بستر هنوز در فاز بتا است و نسخه نهایی آن منتشر نشده است.

- **Microsoft Azure Functions** بستر بدون سرور متن بازی را ارائه می‌دهد که کد آن در Github موجود است. تابع های ارائه شده توسط توسعه دهنده با HTTP webhook که با سرویس های بستر ادغام شده است اجرا می‌شوند. این بستر نیز از زبان های محدودی پشتیبانی می‌کند.
- **OpenLambda** بستر بدون سرور و متن باز جدیدی است که کد آن در Github موجود است. این بستر در مواردی مشکل دارد که توسعه دهندگان در حال رفع آن هستند [۹]:

۱. راه اندازی بسیار آهسته runtime های زبان های برنامه نویسی

۲. استقرار حجم زیادی از کد

۳. پشتیبانی از حفظ وضعیت که با تابع های فاقد وضعیت پیاده شده باشد

۴. استفاده از پایگاه داده ها

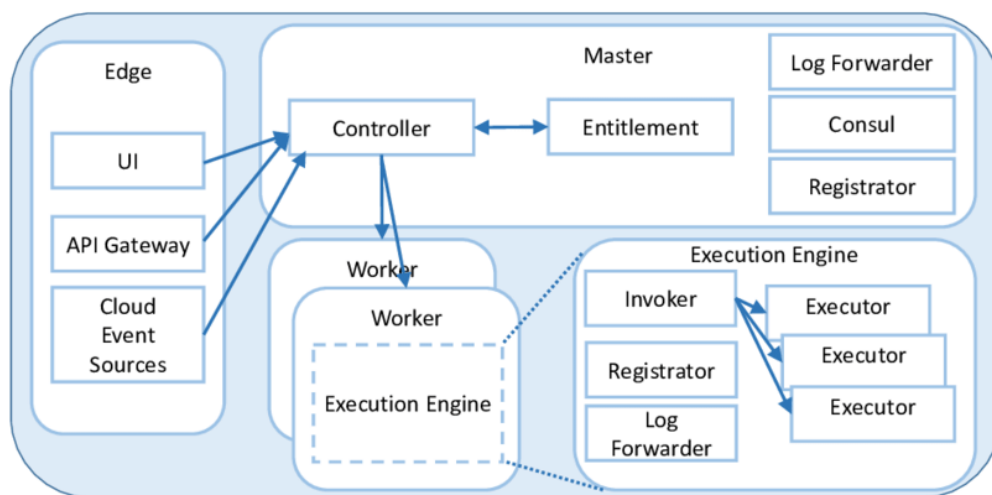
- **Kubeless** این بستر بدون سرور، با راه اندازی بر روی بستر کوبرنتیز، بدون نیاز به پیچیدگی های چارچوب بستر بدون سرور مانند سیستم صف پیام، آن را به یک بستر اجرا کننده ی تابع تبدیل می‌کند. اما این بستر هنوز به اندازه کافی بالغ نشده است که در مرحله تولید مورد استفاده قرار

گیرد. همچنین جامعه‌ی آن به اندازه کافی بزرگ نیست، اسناد آن ناقص هستند و اشکالاتی در آن مشاهده شده است. این بستر نیز متن باز است [۶].

• **OpenFaaS** یکی از بهترین و معروف‌ترین بستر بدون سرور متن باز با استفاده‌ی آسان است. تابع‌ها در بستر OpenFaaS باید از جنس کانینر باشند. به همین دلیل این بستر از تمامی زبان‌های برنامه نویسی پشتیبانی می‌کند. معماری این بستر نسبتاً ساده است و بر روی هماهنگ کننده‌های ابری کوبرنتیز و Docker Swarm راه‌اندازی می‌شود [۱۰].

• **OpenWhisk** کلیدی ترین ویژگی این بستر این است که می‌توان با اتصال تابع‌ها با یکدیگر ترکیبی از تابع‌ها را ساخت و آن را به عنوان تابع جدید ارائه داد. این بستر نیز به صورت متن باز در Github موجود است و از زبان‌های Node.js، Java، Swift و Python پشتیبانی می‌کند. همچنین این قابلیت نیز وجود دارد که تابع‌ها به صورت باینری پیاده‌سازی شده و در یک کانینر قرار گرفته باشند.

این بستر در مقایسه با معماری ساده بدون سرور (شکل ۲-۴)، دارای اجزای مهمی است که امنیت، گزارش گرفتن و نظارت را کنترل می‌کنند. این معماری جدید در شکل ۲-۶ قابل مشاهده است.



شکل ۲-۶: معماری بستر بدون سرور OpenWhisk

در انتها به این نتیجه می‌رسیم که بهترین و پایدارترین راه حل AWS Lambda است. پیشگام بودن در این زمینه باعث شده است تا وقت کافی برای رشد و پشتیبانی از زبان‌های معروف را داشته باشد. بعد از آن Microsoft Azure Functions برای انتخاب مناسب است. به دلیل در دسترس نبودن این سرویس‌ها در داخل کشور، پیاده‌سازی این بستر با بهره‌گیری از پروژه‌های متن باز معقول به نظر

می‌رسد و ارائه چنین سرویس‌هایی در حال آغاز است. با توجه به بررسی‌ها و مقایسه‌های صورت گرفته بهترین راه حل متن باز استفاده از بستر OpenWhisk است که ما در این پروژه نیز از این بستر استفاده کردیم. در ادامه پس از آشنایی با مفاهیم و معماری میکروسرویس، بیشتر با نحوه کارکرد و اجزای مختلف این بستر بدون سرور آشنا خواهیم شد.

۴-۲ میکروسرویس و کانتینرها

معماری میکروسرویس^{۱۲} به مرور در حال کسب محبوبیت فزاینده‌ای است و امروزه تقریباً در همه پروژه‌های نرم‌افزاری عمده از آن استفاده می‌شود. دلیل اصلی این مسئله ناشی از مزیت‌های آن و مسائلی است که حل می‌کند. در این بخش ابتدا به معرفی مفهوم معماری میکروسرویس و شناخت مزیت‌های استفاده از آن می‌پردازیم و سپس فناوری‌های مورد نیاز برای پیاده‌سازی این نوع معماری را بررسی می‌کنیم.

۱-۴-۲ میکروسرویس

در ابتدا باید مفهوم خود میکروسرویس را درک کنیم. میکروسرویس، همان طور که از نام آن مشخص می‌شود، اساساً به سرویس‌های نرم‌افزاری مستقلی گفته می‌شود که کارکردهای تجاری خاصی را در یک برنامه‌ی کاربردی نرم‌افزاری ارائه می‌کنند. این سرویس‌ها می‌توانند به صورت مستقل از هم نگهداری، نظارت و توزیع شوند.

معماری مبتنی بر سرویس^{۱۳} به برنامه‌های کاربردی امکان ارتباط با یکدیگر روی یک رایانه منفرد و یا در زمان توزیع برنامه‌های کاربردی روی چندین رایانه در یک شبکه را ارائه می‌کند. هر میکروسرویس ارتباط اندکی با سرویس‌های دیگر دارد. این سرویس‌ها خودکفا هستند و یک کارکرد منفرد (یا گروهی از کارکردهای مشترک) را ارائه می‌کنند.

معماری میکروسرویس‌ها به طور طبیعی در سازمان‌های بزرگ و پیچیده استفاده می‌شود که در آن‌ها چند تیم توسعه می‌توانند مستقل از هم برای ارائه یک کارکرد تجاری کار بکنند و یا برنامه‌های کاربردی ملزم به ارائه خدمات به یک حوزه تجاری باشند.

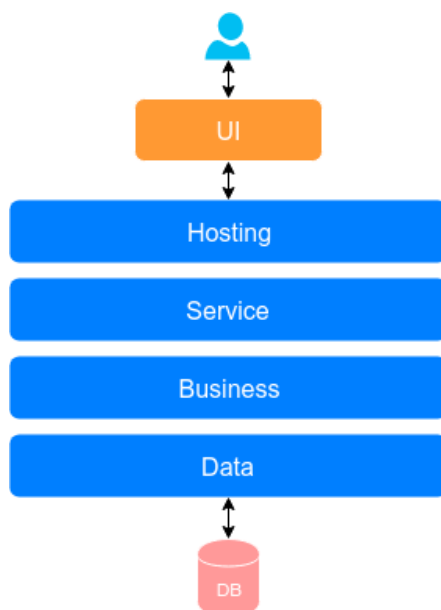
به طور خلاصه یک مفهوم وجود دارد و آن این است که ما همواره ملزم هستیم نرم‌افزار را نگهداری و به‌روزرسانی کنیم. باید فرایند بهبود برنامه‌های کاربردی را آسان‌تر و مقدار زمان مورد نیاز برای ارائه نسخه‌های جدید آن‌ها را کوتاه‌تر کنیم.

معماری یکپارچه

در سال‌های اخیر برنامه‌های کاربردی نرم‌افزاری در معماری‌های چندلایه و یکپارچه پیاده‌سازی می‌شدند. به عنوان نمونه در شکل ۲-۷ یک معماری معمول برای برنامه‌ی کاربردی یکپارچه ارائه شده است:

¹²microservices architecture

¹³Services Oriented Architecture (SOA)



شکل ۲-۷: معماری یکپارچه

در این معماری هر لایه در کد نرم‌افزاری پیاده‌سازی می‌شد و از چندین کلاس و واسط تشکیل شده بود:

لایه داده

این لایه برای ذخیره‌سازی داده‌ها در پایگاه داده و فایل‌ها پیاده‌سازی شده است. تنها مسئولیت این لایه ارائه داده‌ها از منابع داده‌ای مختلف است.

لایه تجاری

مسئولیت لایه تجاری، بازیابی داده‌ها از لایه داده و اجرای محاسبات است. لایه تجاری نمی‌داند که داده‌ها در فایل یا پایگاه داده یا در موارد دیگر هستند. لایه تجاری به لایه داده‌ها وابسته است.

لایه سرویس

لایه سرویس روی لایه تجاری قرار می‌گیرد. این لایه یک پوشش برای لایه تجاری ایجاد می‌کند که شامل امنیت، گزارش‌گیری و وساطت برای فراخوانی کارکردها است. لایه سرویس به لایه تجاری وابسته است.

لایه میزبانی

سرویس‌ها از طریق این لایه میزبانی می‌شوند. لایه میزبانی از فناوری‌های مبتنی بر سرویس مانند REST API برای میزبانی با استفاده از پروتکل‌های مختلف مانند HTTP، HTTPS، TCP و غیره بهره می‌گیرد. کل برنامه‌ی کاربردی به صورت یک پردازش نرم‌افزاری اجرا می‌شود. برنامه‌ها از طریق URL مانند <https://example.com/myapplication> در دسترس هستند.

لایه رابط کاربری

لایه رابط کاربری شامل کدهایی است که در لایه میزبانی مورد ارجاع قرار می‌گیرند و به منظور ایجاد امکان تعامل با برنامه‌ی کاربردی برای کاربران طراحی شده است.

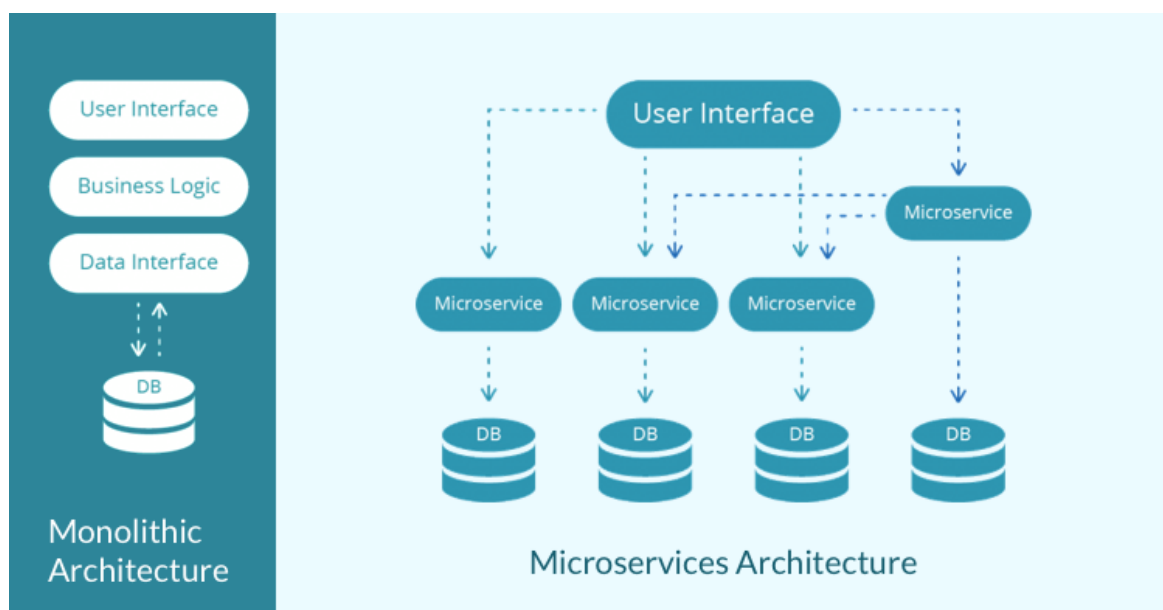
این طراحی به توسعه‌دهندگان امکان می‌دهد که روی یک کارکرد خاص متمرکز شوند، ویژگی را تست کنند، و یا برنامه‌ی کاربردی را با استفاده از کنترل معکوس از طریق تزریق وابستگی‌ها و میزبانی یک برنامه‌ی کاربردی روی ماشین‌های متعدد تجزیه کنند.

طراحی یکپارچه لایه‌بندی شده مزایای زیادی دارد؛ اما نواقصی نیز دارد. در ادامه فهرستی از مشکلات رایج این نوع معماری را مورد اشاره قرار داده‌ایم:

- این طراحی برای مقیاس‌بندی و نگهداری برنامه‌ی کاربردی سراسر نیست. این طراحی زمان مورد نیاز برای قرار دادن کارکردهای جدید در اختیار کاربر را افزایش داده است، چون چرخه توسعه زمان بیشتری طول می‌کشد.
- از آنجا که کل برنامه‌ی کاربردی به صورت یک پردازش منفرد میزبانی می‌شود، هر بار که لازم باشد یک به‌روزرسانی اجرا شود، کل برنامه‌ی کاربردی باید متوقف شود و سپس نسخه جدیدی از آن باید توزیع شود.
- برای ایجاد تعادل در بار کاری، کل برنامه‌ی کاربردی روی چند ماشین توزیع می‌شود. به علاوه، امکان توزیع کارکردهای خاص روی سرورهای چندگانه برای متوازن‌سازی بار وجود ندارد.
- طراحی برنامه‌ی کاربردی پیچیده است، چون همه ویژگی‌ها در یک برنامه‌ی یکپارچه منفرد پیاده‌سازی شده‌اند.
- زمانی که تعداد برنامه‌های کاربردی در سازمان افزایش می‌یابد، توزیع برنامه‌های یکپارچه نیازمند اطلاع‌رسانی و هماهنگی با همه تیم‌های توسعه ویژگی‌های جدید است. این امر موجب افزایش زمان مورد نیاز برای تست و توزیع برنامه می‌شود.
- این معماری برنامه‌ی کاربردی را وادار می‌کند که در یک مجموعه فناوری منفرد پیاده‌سازی شود.
- در مواردی که زمان تحویل طولانی‌تر شود، در طی زمان نیازمند پول بیشتری برای توسعه و نگهداری برنامه‌ی کاربردی خواهد بود.
- از آنجا که همه کد درون یک برنامه‌ی کاربردی منفرد قرار دارد، نگهداری کد پس از مدتی به سرعت دشوار می‌شود.

معماری میکروسرویس

میکروسرویس‌ها برای حل مسائل اشاره شده فوق معرفی شده‌اند. معماری میکروسرویس یک بهینه‌سازی در زمینه معماری یکپارچه محسوب می‌شود. در این معماری هر کارکرد تجاری به صورت یک سرویس ارائه می‌شود. هر سرویس می‌تواند به صورت مستقل از سرویس‌های دیگر میزبانی و توزیع شود. در شکل ۸-۲ معماری یکپارچه با معماری میکروسرویس مقایسه شده است.

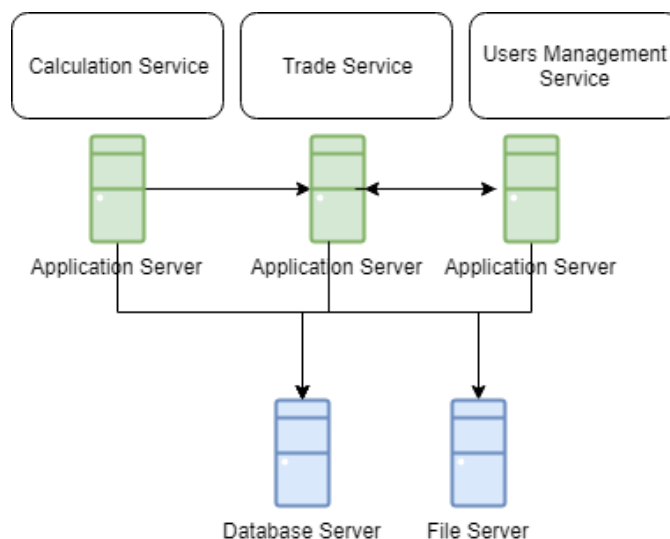


شکل ۸-۲: مقایسه معماری میکروسرویس با معماری یکپارچه

- هر سرویس را می‌توان یک برنامه‌ی کاربردی کوچک تصور کرد.
- همه سرویس‌ها می‌توانند حتی زمانی که سرویس‌ها روی ماشین‌های مختلف هستند، با همدیگر ارتباط داشته باشند. این وضعیت در ادامه امکان پیاده‌سازی کارکردهای جدید در سرویس‌ها را فراهم می‌سازد.
- برنامه‌های کاربردی در نهایت بسیار پایدارتر می‌شوند، چون هر ویژگی می‌تواند به صورت مستقل تست و توزیع شود.
- از آنجا که هر سرویس روی پردازش مجزایی میزبانی می‌شود، اگر یک سرویس به نقطه تنگنای برنامه‌ی کاربردی تبدیل شود و به منابع زیادی نیاز داشته باشد، در این صورت می‌توان آن را بدون هیچ گونه تأثیر سوء روی سرویس‌های دیگر، به ماشین دیگری انتقال داد.

- زمانی که کاربران بیشتری شروع به استفاده از یک ویژگی برنامه‌ی کاربردی بکنند، آن سرویس می‌تواند با توزیع روی رایانه‌های قدرتمندتر یا از طریق استفاده از کش بدون این که روی سرویس‌های دیگر هیچ تأثیری بگذارد، مقیاس‌بندی شود.
 - این معماری پایداری برنامه‌ی کاربردی را نیز افزایش می‌دهد، چون هر سرویس می‌تواند به صورت مستقل ساخته، تست، توزیع و استفاده شود.
 - کد برنامه‌ی کاربردی می‌تواند به سادگی نگهداری شود و پردازش‌ها می‌توانند به صورت مجزا تحت نظارت قرار بگیرند.
 - توسعه‌دهندگان اختصاصی می‌توانند سرویس‌ها را به صورت مستقل از هم پیاده‌سازی کرده و این سرویس‌ها را بدون تأثیرگذاری روی سرویس‌های دیگر انتشار دهند.
 - بدین ترتیب نقطه شکست منفرد نیز از بین می‌رود، زیرا یک سرویس می‌تواند بدون تأثیرگذاری روی ویژگی‌های دیگری که برنامه‌ی کاربردی نرم‌افزاری ارائه می‌کند، متوقف شود.
 - در نتیجه این طراحی زمان مورد نیاز برای تحویل نسخه‌های جدید را کاهش می‌دهد و بنابراین هزینه را در طی زمان کاهش می‌دهد.
 - قابلیت استفاده مجدد از کد افزایش می‌یابد، زیرا یک ویژگی به صورت سرویس میزبانی شده است و امکان استفاده چند سرویس از یک ویژگی به جای پیاده‌سازی مجدد کد در هر مورد وجود دارد.
 - معماری مبتنی بر سرویس امکان استفاده از مجموعه متنوعی از فناوری‌ها برای رفع نیازها را دارد. به عنوان نمونه بسته‌های تحلیل داده زبان پایتون می‌توانند به صورت مجزا توزیع و میزبانی شوند و همزمان می‌توان از NodeJS برای پیاده‌سازی سرویس‌ها استفاده کرد و ReactJS نیز برای پیاده‌سازی رابط کاربری مورد استفاده قرار گیرد.
- در شکل ۲-۹ نشان داده‌ایم که سرویس‌های متفاوت چگونه می‌توانند روی سرورهای مختلف برنامه‌ی کاربردی توزیع شوند [۱۳].
- میکروسرویس‌ها می‌توانند با استفاده از طیف متنوعی از فناوری‌ها از جمله داکر و کوبرنتیز پیاده‌سازی و میزبانی شوند. در ادامه به معرفی این فناوری‌ها می‌پردازیم. اما در ابتدا نیاز به آشنایی با مفهوم کانتینر^{۱۴} داریم.

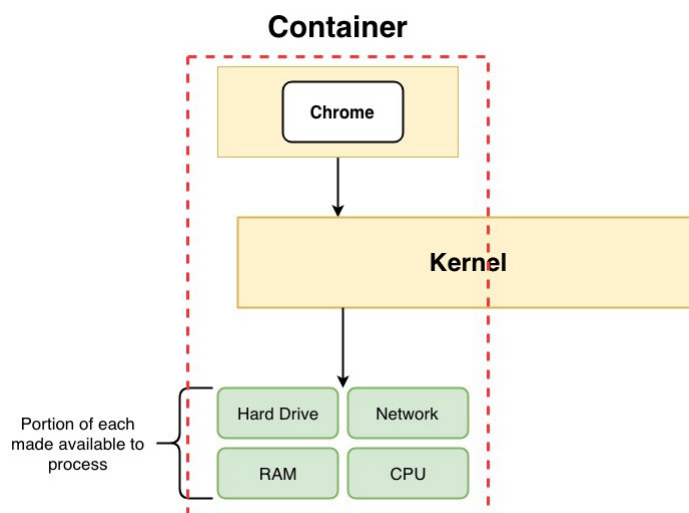
¹⁴container



شکل ۲-۹: توزیع میکروسرویس‌ها روی سرورهای مختلف

۲-۴-۲ کانتینرها

کانتینرها اجازه بسته‌بندی برنامه‌های کاربردی را به همراه تمام قسمت‌هایی که نیاز دارند شامل کتابخانه‌ها و دیگر وابستگی‌هایش، به توسعه‌دهنده‌ها می‌دهند. به عبارت دیگر می‌توانیم کانتینر را شامل پردازنده^{۱۵} در حال اجرا و منابع سخت‌افزاری اختصاص داده شده به آن پردازنده تعریف کرد (شکل ۲-۱۰). پس کانتینر به معنی یک قسمت سخت‌افزاری و قابل رویت نیست بلکه یک تعریف انتزاعی به معنای یک پردازنده یا گروهی از پردازنده‌ها به همراه منابع تخصیص داده شده به آن‌ها می‌باشد.



شکل ۲-۱۰: نمایش یک کانتینر

¹⁵process

کانتینرها مکانیزم منطقی بسته‌بندی ارائه می‌دهند بدین معنا که برنامه‌های کاربردی می‌توانند از محیطی که در آن در حال اجرا هستند جدا شوند. این جداسازی باعث می‌شود تا برنامه‌های کاربردی مبتنی بر کانتینرها به راحتی و بدون درنظر گرفتن محیط هدف مستقر شوند. کانتینرایز کردن برنامه‌ها منجر به آن می‌شود که توسعه‌دهندگان تنها به منطق برنامه فکر کنند و تیم‌های عملیات تکنولوژی اطلاعات بر روی پیاده‌سازی و مدیریت آن تمرکز کنند.

برخلاف یک ماشین مجازی، یک کانتینر دارای یک سیستم عامل مجزا نمی‌باشد. سیستم عامل‌ها kernel دارند که یک پروسه اجرایی نرم‌افزاری^{۱۶} می‌باشد و وظیفه آن کنترل دسترسی بین برنامه‌های در حال اجرا به منابع سخت‌افزاری موجود در کامپیوتر است.

سیستم عامل‌ها ویژگی به نام name spacing دارند که به وسیله آن می‌شود قطعه‌هایی^{۱۷} از منابع سخت‌افزاری موجود ساخت و آنها را به یک برنامه خاص اختصاص داد. در نتیجه به وسیله name spacing می‌شود منبع سخت‌افزاری را بر طبق پرازنه یا برنامه اجرایی جدا و ایزوله کرد. البته ویژگی name spacing فقط برای سخت‌افزار نمی‌باشد و برای نرم‌افزار هم تعریف می‌شود. یک ویژگی دیگر هم با نام control group وجود دارد که به وسیله آن می‌توان مقدار منابعی که یک پرازنه می‌تواند استفاده کند را محدود کرد. در نتیجه به وسیله این دو ویژگی می‌توان برای هر پرازنه منابعی را اختصاص داد و آن منابع را محدود کرد.

در این میان نیاز به تعریف مفهومی با عنوان ایمج^{۱۸} داریم. ایمج فایل‌های سیستمی^{۱۹} است که شامل همه‌ی وابستگی‌ها و تنظیمات موردنیاز برای اجرای یک برنامه خاص می‌باشد. ایمج را می‌توان به کانتینر تبدیل کرد. برای این کار ابتدا kernel منابع سخت‌افزاری لازم را جدا می‌کند به طوری که فقط پرازنه‌های کانتینر موردنظر می‌توانند به این منابع دسترسی داشته باشند. در ادامه فایل‌های موردنیاز آن کانتینر نیز جدا می‌شوند و بدین ترتیب یک کانتینر ساخته می‌شود.

۳-۴-۲ داکر

در قسمت قبل با مفهوم کانتینر آشنا شدیم. حال سوال این است که چگونه می‌توانیم کانتینر را بسازیم و سپس با آن کار کنیم؟ در این قسمت با ابزاری به نام داکر^{۲۰} آشنا می‌شویم. داکر وظیفه مدیریت کانتینرها را به عهده دارد و بیشتر شبیه یک ماشین مجازی عمل می‌کند، تفاوت داکر با ماشین مجازی در این است که در ماشین مجازی برای اجرای برنامه‌ی کاربردی و برنامه‌های

^{۱۶}running software process

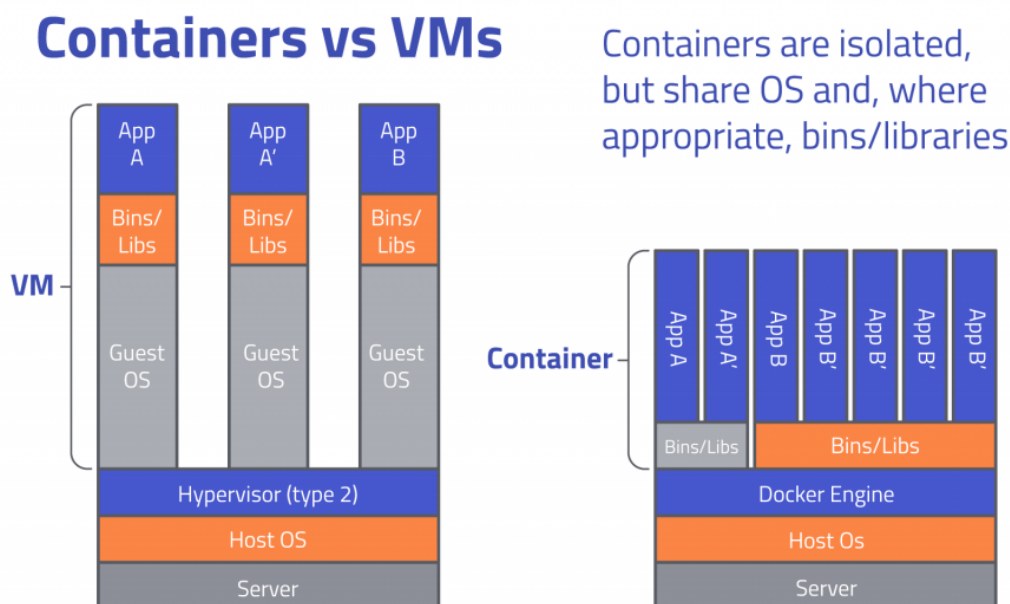
^{۱۷}segments

^{۱۸}image

^{۱۹}file system

^{۲۰}Docker

مختلف که بخواهیم به صورت ایزوله و مجزا از هم کار کنند باید ماشین‌های مجازی مختلف ساخته شود که همین موضوع بار پردازشی و هدر رفت منابع سیستمی را روی سرور به همراه دارد. ولی در داکر روی یک ماشین مجازی خاص که می‌تواند دارای سیستم عامل ویندوز یا لینوکس باشد، ماژول داکر نصب شده و سپس روی سرویس داکر، کانتینرهای مختلف حاوی برنامه‌های کاربردی مختلف نصب و اجرا می‌شوند بدون اینکه کانتینرها به هم دسترسی داشته باشند. (شکل ۲-۱۱) بدین صورت کانتینرها از هم ایزوله هستند و نیاز برای ایجاد چندین ماشین مجازی را مرتفع می‌سازند. از طرف دیگر کانتینرهای داکر به راحتی در فضای رایانش ابری قرار می‌گیرد و به نوعی طراحی شده که تقریباً تمامی برنامه‌های کاربردی که از متدولوژی توسعه عملیات^{۲۱} بهره می‌گیرند با داکر تعامل داشته باشند.



شکل ۲-۱۱: مقایسه‌ی داکر و ماشین مجازی

داکر محیطی محلی را برای توسعه فراهم می‌کند که این محیط دقیقاً عمل کرد یک سرور را در اختیار توسعه‌دهندگان قرار می‌دهد. این امر برای روش توسعه یکپارچه‌سازی و تحویل مداوم (CI/CD)^{۲۲} کاربرد زیادی دارد. از این طریق می‌توان چندین محیط توسعه را از یک میزبان مشخص با یک نرم‌افزار، سیستم‌عامل و تنظیمات واحد اجرا کرد. از طرف دیگر پروژه را می‌توان روی چند سرور جدید و مختلف آزمایش کرد و تمامی اعضای گروه بر روی یک پروژه واحد با تنظیمات همانند قادر به همکاری هستند.

^{۲۱} توسعه عملیات (DevOps)، مجموعه‌ای از روشها، فرایندها و ابزارهایی است که با تمرکز بر همکاری و یکپارچگی بین تیم‌های توسعه نرم‌افزار و عملیات فناوری اطلاعات، ارزش‌های تولید شده را به‌طور سریع و مداوم به مشتریان نهایی می‌رساند.

این کار توسعه‌دهندگان را قادر می‌سازد تا نسخه‌های جدید برنامه خود را به سرعت آزمایش کرده و از عملکرد صحیح آن اطمینان خاطر حاصل کنند.

تمامی زیرساخت‌های فناوری اطلاعات نیازمند مدیریت و نظارت هستند و در همین راستا کانتینرها نیز باید نظارت شوند و در حالت کنترل‌شده‌ای قرار بگیرند. در غیر این صورت مشخص نخواهد شد سرور چه برنامه‌هایی را اجرا می‌کند. خوشبختانه از برنامه‌های دواپس می‌توان برای مانیتور کانتینرهای داکر کمک گرفت، اما به این نکته نیز باید اشاره کرد که این برنامه‌ها برای کانتینرها بهینه نشده‌اند. اینجاست که باید سراغ ابزارهای مدیریت و نظارت رایانش ابری را گرفت. ابزارهایی مانند Docker، Mesosphere و Kubernetes، Swarm در این زمینه گزینه‌های خوبی به نظر می‌رسند و تجربه نشان داده است در بین این ابزارها Kubernetes محبوبیت بیشتری پیدا کرده است. [۱۸]

۴-۴-۲ کورنتیز

مفاهیم اولیه

کورنتیز یک هماهنگ‌کننده‌ی کانتینری^{۲۳} متن باز، قابل حمل و قابل توسعه می‌باشد که برای اجرا و مدیریت کانتینرها و همچنین جهت تسهیل تنظیمات توسعه داده شده‌است. از ویژگی‌های کورنتیز می‌توان به موارد زیر اشاره کرد:

۱. کشف سرویس و توزیع بار^{۲۴}:

کورنتیز بدون احتیاج به تغییر برنامه کاربردی این امکان را می‌دهد تا سرویس خود را ثبت نام کنید و اجازه دهید تا بقیه سرویس‌ها از وجود آن خبردار شوند. کورنتیز به هر کانتینر یک آدرس آی‌پی اختصاصی می‌دهد و برای هر گروه از کانتینرها یک نام DNS اختصاص می‌دهد که امکان توزیع بار بین آن‌ها را فراهم می‌کند.

۲. هدایت کردن انبارهای ذخیره اطلاعات^{۲۵}:

کورنتیز این اجازه را می‌دهد تا به صورت خودکار انبارهای ذخیره اطلاعات سیستم را مقیاس‌دهی کنید.

۳. تطبیق خودکار^{۲۶}:

بدین معنی که حالت مورد انتظار خود را به کورنتیز معرفی می‌کنید و کورنتیز حالت فعلی را به

²³ container orchestration

²⁴ Service discovery and load balancing

²⁵ Storage orchestration

²⁶ Automated rollouts and rollbacks

حالت مورد انتظار تبدیل می‌کند.

۴. بسته‌بندی خودکار^{۲۷}:

با توجه به نیازمندی و منابع موردنیاز هر کانتینر، کوبرنتیز به صورت اتوماتیک کانتینرها را بر روی نودها قرار می‌دهد.

۵. ترمیم خودکار^{۲۸}:

کوبرنتیز به صورت خودکار تمامی کانتینرهایی که موفق به اجرا نشوند را بازنشانی می‌کند. همچنین در صورتی که نودی که کانتینر بر روی آن در حال اجرا است، دچار مشکل شود، کوبرنتیز تمامی کانتینرهای در حال اجرا بر روی آن را مجدداً روی نودهای باقی‌مانده قرار می‌دهد.

۶. توسعه‌پذیری افقی:

کوبرنتیز این امکان را می‌دهد تا برنامه کاربردی خود را به صورت اتوماتیک کوچک یا بزرگ کنید. همچنین این امکان را می‌دهد تا این کار را بر عهده‌ی خود کوبرنتیز قرار دهید و این کار به صورت خودکار انجام گیرد.

۷. مدیریت secret ها و تنظیمات^{۲۹}:

کوبرنتیز امکان ذخیره و مدیریت اطلاعات حساس مانند رمزعبورها، OAuth tokens ، ssh keys را می‌دهد. می‌توانید بدون بازسازی کانتینرها، secrets و تنظیمات، محصول کاربردی را پیاده‌سازی و یا به‌روز کنید. این کار بدون نمایش دادن secret در فایل‌های تنظیمات انجام می‌گیرد.

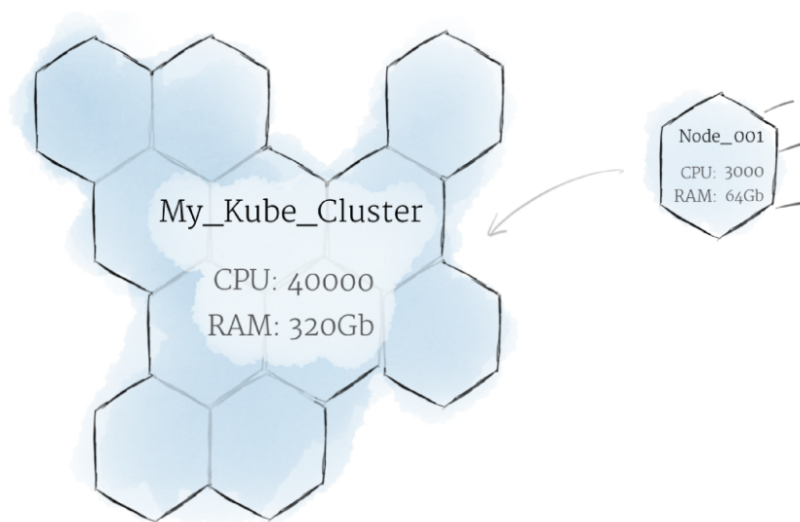
²⁷Automatic bin packing

²⁸Self-healing

²⁹Secret and configuration management

معماری کوبرنتیز

اولین مرحله جهت کار کردن با کوبرنتیز شناخت معماری و اجزای آن است. یک نود کوچک‌ترین جزء سخت‌افزارهای محاسباتی در کوبرنتیز است. نود نماینده یک ماشین در خوشه است. در بیشتر سیستم‌های تولید، به احتمال زیاد نود یک ماشین فیزیکی در مرکز داده و یا یک ماشین مجازی بر روی ارائه‌دهندگان ابری می‌باشد. تعریف یک نود به عنوان یک ماشین، این اجازه را می‌دهد که یک لایه از مفاهیم را استفاده کنیم. بدین معنا که نیاز نیست نگران مشخصات یکتا ماشین باشیم بلکه هر ماشین را مجموعه‌ای منابع مانند CPU و GPU در نظر می‌گیریم که می‌توانند استفاده شوند. بدین ترتیب هر ماشین می‌تواند جایگزین ماشینی دیگر در دنیای کوبرنتیز شود. اگرچه کار کردن با تکنودها می‌تواند مفید باشد اما این روش کوبرنتیز نیست. به طور کلی در کوبرنتیز ما یک خوشه^{۳۰} را به جای حالت‌های تکنودها در نظر می‌گیریم (شکل ۲-۱۲). در کوبرنتیز نودها منابع خود را به اشتراک می‌گذارند تا یک ماشین قدرتمندتر شکل گیرد. وقتی برنامه‌های خود را بر روی خوشه مستقر می‌کنید، کارهای اجرایی برنامه به صورت هوشمندانه بر روی تک نودها توزیع می‌شود. اگر نودی اضافه یا حذف شود، خوشه کارها را بر روی نودها جابه‌جا می‌کند. لازم به ذکر است برنامه و برنامه‌نویس با نحوه اجرای برنامه و توزیع آن بر روی نودها درگیر نیست.



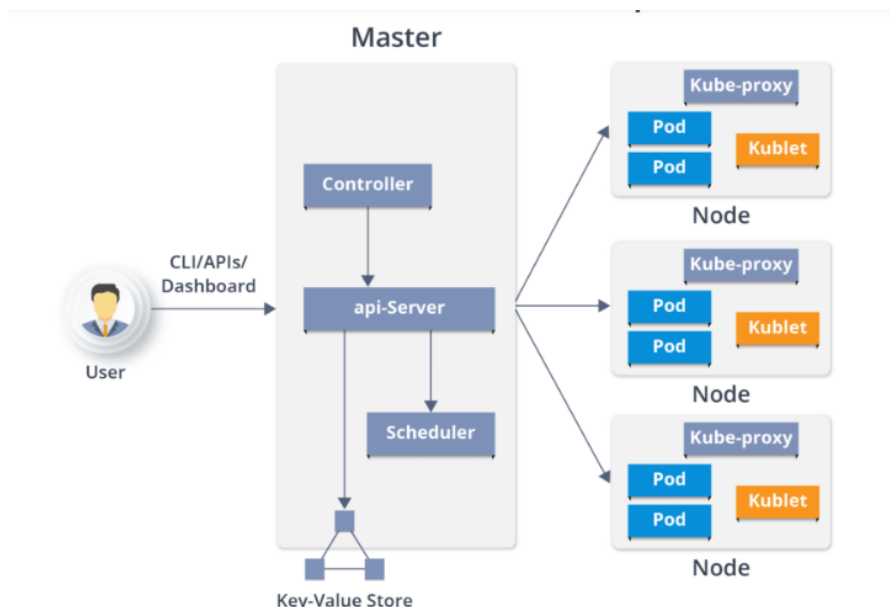
شکل ۲-۱۲: رابطه‌ی نود و خوشه در کوبرنتیز

تصویر (شکل ۲-۱۳) معماری یک خوشه کوبرنتیز^{۳۱} می‌باشد. از آن جایی که کوبرنتیز یک خوشه محاسباتی را پیاده می‌کند، همه چیز در داخل این خوشه کوبرنتیز اتفاق می‌افتد. این خوشه میزبانی

^{۳۰}cluster

^{۳۱}Kubernetes Cluster

یک نود با نام مستر^{۳۲} و نودهای دیگر با نام نود کارگر^{۳۳} را بر عهده دارد. همچنین خارج خوشه یک متعادل کننده بار^{۳۴} وجود دارد که ترافیک خارجی را کنترل و بین نودها پخش می کند. مستر کنترل خوشه و نودهای داخل آن را برعهده دارد. هر نود میزبان یک یا تعدادی پاد^{۳۵} می باشد و هر پاد شامل گروهی از کانتیرها می باشد که برای یک هدف با یکدیگر در تعامل هستند.



شکل ۲-۱۳: معماری خوشه کوبرنتیز

در نتیجه به طور کلی معماری خوشه‌ی کوبرنتیز دارای اجزا اصلی زیر می باشد:

۱. نودهای مستر

۲. نودهای کارگر

۳. مراکز ذخیره اطلاعات توزیع شده به صورت key-value^{۳۶}

نود مستر

نود مستر را می توان یک نقطه شروع برای همه کارهای اجرایی که قرار است آن خوشه کوبرنتیز بر عهده داشته باشد، دانست. می توان برای چک کردن تحمل خطا بیش از یک نود مستر را در خوشه قرار

^{۳۲}master

^{۳۳}worker node

^{۳۴}load balancer

^{۳۵}pod

^{۳۶}Distributed key-value store(etcd)

داد. در این حالت نیز یکی از نودهای مستر به عنوان نود اصلی شناخته می‌شود و دستورات لازم را به آن اعمال می‌کنیم. تنها نودهای مستر توانایی اجرای عملیات زمان‌بندی کانتینرها بر روی نودهای کارگر یا خودشان را دارا می‌باشند. هر نود مستر یک رابط برنامه‌نویسی دارد که از طریق آن با رابط کنسولی و یا رابط تحت وب می‌توان با نود مستر ارتباط برقرار نمود.

در کوبرنتیز نودها به وسیله مستر ساخته و مدیریت می‌شوند. در واقع در مستر برنامه‌هایی اجرا می‌شوند، که این برنامه‌ها ساخت و اجرای نودها را کنترل می‌کنند. توسعه‌دهندگان برنامه‌های خود را در قالب فایل تنظیمات^{۳۷} به مستر می‌دهند و مستر برنامه خواسته‌شده را در نودها پخش می‌کند.

نود مستر دارای چهار جز اصلی است که به تشریح آن‌ها می‌پردازیم:

۱. Api server:

تمام کارهای مدیریتی از طریق API server در نود مستر انجام می‌شود. تمام دستورات REST که به API server فرستاده می‌شوند در این قسمت اعتبارسنجی و پردازش می‌شوند. پس از اجرای درخواست، نتیجه به دست‌آمده در مراکز ذخیره اطلاعات توزیع شده ذخیره می‌شوند.

۲. زمان‌بند^{۳۸}:

این قسمت وظیفه زمان‌بندی کارها را برای نودهای کارگر بر عهده دارد. همچنین اطلاعات مصرف منابع هر نود کارگر را در خود نگه می‌دارد.

۳. مدیر کنترل^{۳۹}:

یک daemon^{۴۰} است که وظیفه کنترل حلقه‌های بدون انتها را بر عهده دارد. همچنین وظیفه اجرای توابع چرخه زندگی^{۴۱} مانند ساخت namespace و lifecycle، جمع‌آوری زباله (باقی‌مانده) رویدادها، جمع‌آوری زباله (باقی‌مانده) پادهای خاتمه‌یافته، جمع‌آوری زباله (باقی‌مانده) پاک‌کردن‌های آبشاری، جمع‌آوری زباله (باقی‌مانده) نودها و ... را بر عهده دارد. از وظایف دیگر این قسمت نظارت و دنبال کردن وضعیت مورد انتظار اشیاء و وضعیت فعلی اشیاء می‌باشد و زمانی که این دو وضعیت در شرایط برابر نباشند، حلقه اصلاحی را به سیستم اعمال می‌کند تا این دو وضعیت برابر شوند.

^{۳۷}config file

^{۳۸}scheduler

^{۳۹}controller manager

^{۴۰}در سیستم عامل‌ها، یک برنامه کامپیوتری است که به جای آن که تحت کنترل مستقیم یک کاربر باشد، به عنوان یک پردازنده در پس‌زمینه اجرا می‌شود.

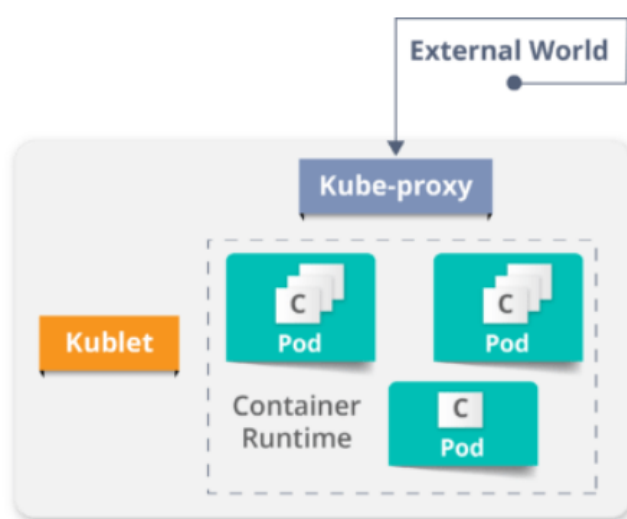
^{۴۱}lifecycle

۴. ETCD:

یک مرکز ذخیره‌سازی توزیع‌شده به صورت key-value می‌باشد که حالت خوشه را در خود نگه می‌دارد. هم می‌تواند قسمتی از نود مستر باشد و هم می‌تواند یک تنظیمات خارجی باشد. علاوه بر حالت خوشه جزییات تنظیمات خوشه را نیز در خود نگه‌داری می‌کند.

نود کارگر

یک سرور فیزیکی و یا یک ماشین مجازی می‌باشد که به وسیله پاد برنامه‌های کاربردی را اجرا می‌کند و به وسیله نود مستر کنترل می‌شود. نود کارگر دارای قسمت‌های زیر می‌باشد (شکل ۲-۱۴):



شکل ۲-۱۴: معماری نود کارگر

۱. container runtime:

جهت اجرای چرخه زمان کانتینرها، نود کارگر به این قسمت نیازمند است. گاهی اوقات داکر به عنوان یک container runtime معرفی می‌شود اما در واقع داکر پلتفرمی می‌باشد که از کانتینرها به عنوان یک container runtime استفاده می‌کند.

۲. Kubelet:

یک عامل است که با نود مستر ارتباط برقرار می‌کند و بر روی نودهای کارگر اجرا می‌شود. همچنین مشخصات فنی پادها را دریافت می‌کند و کانتینرهایی که به پادها مرتبط هستند را اجرا می‌کند. سپس از اجرا و سلامتی کانتینرها اطمینان حاصل می‌کند.

۳. Kube-proxy:

جهت ارتباط با شبکه (host sub-netting) بر روی هر نود اجرا می‌شود و اطمینان حاصل می‌کند که سرویس‌ها برای قسمت‌های خارجی در دسترس هستند. همچنین به عنوان یک پروکسی شبکه و یک متعادل کننده بار برای سرویس عمل می‌کند و مسیریابی شبکه را برای بسته‌های TCP و UDP انجام می‌دهد. این پروکسی شبکه دائماً به API server گوش می‌دهد. برای هر سرویس یک مسیریابی انجام می‌دهد تا بتوان به آن دسترسی داشت.

۴. Pod:

شامل یک یا چند کانتینر است که کانتینرها از لحاظ منطقی با هم برابرند. پادها به عنوان یک واحد منطقی اجرا می‌شوند. پادها نیاز ندارند که بر روی یک ماشین مجازی اجرا شوند بلکه می‌توان هر کدام را بر روی یک ماشین مجازی متفاوت اجرا کرد.

اشیاء کوبرنتیز

در فضای یک نود کارگر می‌توان از اشیاء^{۴۲} مختلفی استفاده کرد. اشیاء در کوبرنتیز موجودیت‌های ماندگار هستند. کوبرنتیز از این موجودیت‌ها برای نشان دادن حالت و وضعیت خوشه استفاده می‌کند. مهم‌ترین مواردی که این موجودیت‌ها توصیف می‌کنند شامل مشخص کردن برنامه‌های کاربردی کانتینر شده در حال اجرا، منابع در دسترس برای برنامه‌های در حال اجرا، سیاست‌هایی که مشخص می‌کند این برنامه‌ها چگونه رفتار کنند و ... می‌باشد. اشیاء کوبرنتیز یک بار توسط توسعه‌دهنده ایجاد می‌شوند و سیستم کوبرنتیز دائماً این اشیاء را زیر نظر می‌گیرد و از وجود آن‌ها اطمینان حاصل می‌کند. با ایجاد یک شیء، به سیستم کوبرنتیز می‌گویید که می‌خواهید خوشه چگونه کار کند. بدین ترتیب حالت مورد انتظار خوشه را مشخص می‌کنید. جهت ساخت، اصلاح و یا حذف اشیاء نیازمند kubernetes API هستید. هر شیء کوبرنتیز شامل دو قسمت تودرتو می‌باشد و هر قسمت خود یک شیء است. این دو قسمت تنظیمات شیء را اداره می‌کنند. این دو قسمت spec و status نام دارند. قسمت spec که توسعه‌دهنده باید آن را مهیا کند، حالت مورد انتظار شیء را توصیف می‌کند. قسمت status حالت حقیقی و فعلی شیء را توصیف می‌کند که به وسیله سیستم کوبرنتیز تهیه و به‌روز می‌شود.

⁴²object

تعدادی از اشیاء کوبرنتیز را معرفی می‌کنیم [۱۷]:

۱. Pod:

پاد گروهی از یک یا چند کانتینر با انبار ذخیره‌سازی^{۴۳} و شبکه به‌اشتراک گذاشته شده می‌باشد. کوبرنتیز به طور مستقیم کانتینرها را اجرا نمی‌کند، بلکه یک یا چند کانتینر را در ساختاری با نام پاد بسته‌بندی می‌کند. کانتینرهای داخل یک پاد دارای منابع یکسان به همراه شبکه محلی یکسان هستند و به آسانی با دیگر کانتینرهای داخل پاد ارتباط برقرار می‌کنند. این امر به این دلیل است که بر روی یک ماشین قرار دارند. پاد به عنوان واحدی برای تکثیر در کوبرنتیز شناخته می‌شود. اگر استفاده از برنامه کاربردی به اندازه‌ای شود که یک پاد توانایی تحمل بار آن را نداشته باشد، کوبرنتیز می‌تواند به گونه‌ای تنظیم شده باشد که در این حالت پاد موردنظر را تکثیر کند. حتی اگر برنامه زیر بار سنگین نباشد، حالت استاندارد این است که چند کپی از پاد در یک لحظه در حال اجرا باشد تا تعادل بار و کنترل شکست آسان‌تر شود. پادها توانایی نگه‌داری چندین کانتینر را دارند اما ترجیح بر آن است که در هر پاد یک کانتینر نگه‌داری شود. چرا که پادها واحدی برای مقیاس‌پذیری هستند در نتیجه در صورت زیاد و یا کم شدن تعداد پادها، همه کانتینرهای داخل پاد با هم زیاد و کم می‌شوند درحالی که ممکن است تنها لازم باشد مقیاس یکی از کانتینرها تغییر کند. در نتیجه این امر باعث هدر رفتن منابع و افزایش هزینه‌ها می‌شود.

۲. Replica set:

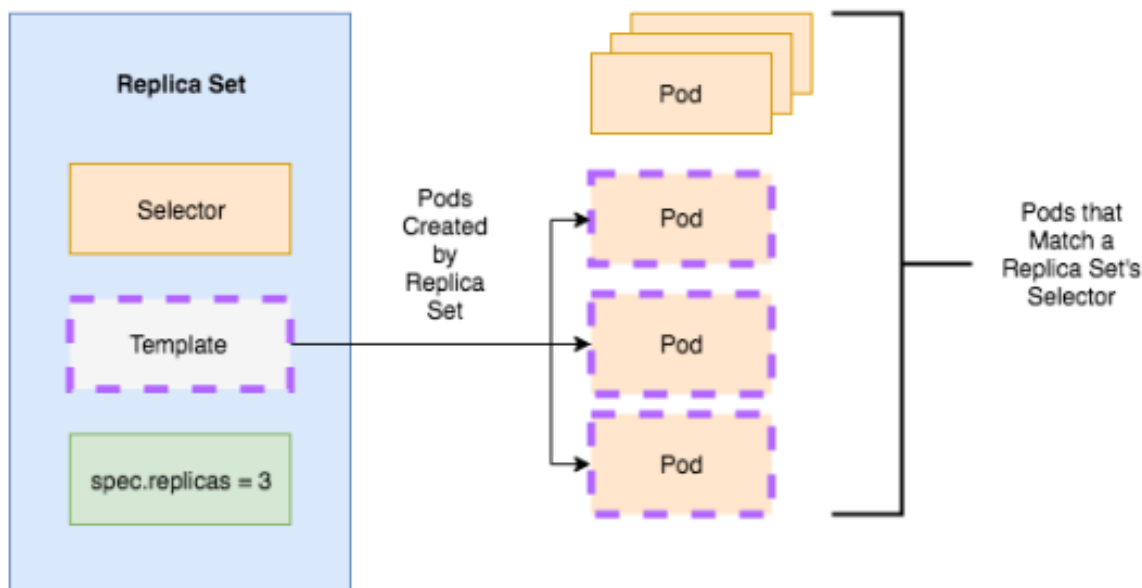
یکی از مزایای کلیدی پادها این است که به توسعه‌دهندگان این اجازه را می‌دهند تا مجموعه‌ای از کانتینرها را به عنوان یک برنامه واحد گروه‌بندی و دسته‌بندی کنند و به آسانی با آن‌ها به عنوان یک حجم کار^{۴۴} کار کنند. پس از ساخت پادها، نمونه‌هایی از پادهای ساخته شده می‌توانند به صورت افقی مقیاس‌بندی^{۴۵} شوند تا برنامه‌های چند کانتینری در دسترس باشند. برای مدیریت مقیاس‌پذیری پادها، کوبرنتیز از یک API object استفاده می‌کند که replica set نام دارد. بر طبق مستندات کوبرنتیز، replica set ها این اطمینان را می‌دهند که تعداد مشخصی از کپی‌های پادها در هر زمان در حال اجرا باشند. در replica set سه قسمت اصلی وجود دارد (شکل ۲-۱۵). قسمت اول selector نام دارد. وظیفه این قسمت انتخاب پاد موردنظر جهت مدیریت آن است. قسمت دوم template نام دارد. replica set با توجه به این قسمت پاد خواسته شده ساخته می‌شود. قسمت سوم replicas نام دارد که تعداد پادهای خواسته شده در این قسمت قرار می‌گیرد. با این

⁴³storage

⁴⁴workloads

⁴⁵scale

که replica set ها توانایی مدیریت پادها را دارند اما قادر به به‌روزرسانی متحرک^{۴۶} نیستند. به همین دلیل از تعریف جدیدی به نام deployment استفاده می‌شود [۱].



شکل ۲-۱۵: نحوه عملکرد replica set

۳. Deployment:

با اینکه پادها اساسی‌ترین واحد محاسباتی در کوبرنتیز می‌باشند، اما به صورت مستقیم بر روی خوشه کوبرنتیز راه‌اندازی نمی‌شوند. در عوض پادها به وسیله‌ی یک لایه انتزاعی به نام deployment مدیریت می‌شوند. هدف اولیه deployment ها این است که مشخص کنند چند کپی از پاد در یک زمان باید در حال اجرا باشند. وقتی یک deployment به خوشه اضافه می‌شود، به صورت اتوماتیک ابتدا تعداد پادهای درخواست‌شده را راه‌اندازی می‌کند سپس بر آن‌ها نظارت می‌کند. اگر یک پاد بمیرد، deployment به صورت خودکار دوباره آن را می‌سازد.

از نگاهی دیگر deployment، پادها و replica set ها را بسته‌بندی می‌کند و روشی اعلامی^{۴۷} برای به‌روزرسانی حالات آن‌ها مهیا می‌کند (شکل ۲-۱۶).

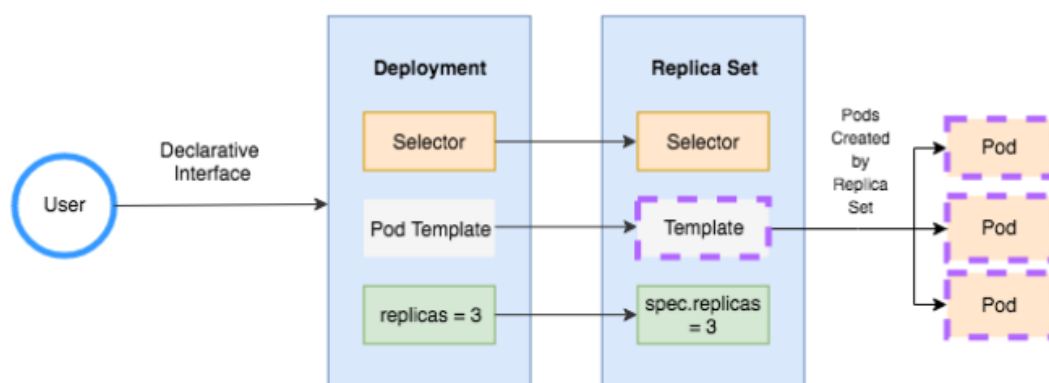
۴. Service:

یک مفهوم است که مجموعه‌ی منطقی از پادها و سیاست‌های دسترسی به آن‌ها را تعریف می‌کند. Service ها یک نقطه پایان^{۴۸} برای پادها مشخص می‌کنند تا بتوانیم به آن‌ها دسترسی داشته

^{۴۶}rolling update

^{۴۷}declarative method

^{۴۸}endpoint



شکل ۲-۱۶: روش اعلامی ایجاد شده توسط deployment

باشیم و بدانیم در چه وضعیتی هستند.

۵. Storage class:

راهی برای توصیف کلاس‌های فضای ذخیره توسط مدیران تهیه می‌کند.

۶. Persistent Volumes:

اگر یک برنامه اطلاعات خود را در یک فایل محلی در نود ذخیره کند، زمانی که برنامه به هر دلیلی به نود دیگر انتقال پیدا می‌کند، تضمینی وجود ندارد که اطلاعات در دسترس باشند. به همین علت می‌توان گفت انبارهای ذخیره‌سازی که در نودها وجود دارند به عنوان حافظه‌های موقت عمل می‌کنند و داده‌هایی در آن‌ها جا می‌گیرند پایدار نیستند. برای ذخیره اطلاعات به صورت دائمی کوبرنتیز از Persistent Volumes استفاده می‌کند. Persistent Volume ها چرخه حیاتی دارند و مستقل از پادهایی هستند که به آن‌ها متصل می‌شوند.

۷. Persistent Volume Claim:

یک تعریف انتزاعی از Persistent Volume می‌باشد. Persistent Volume ها منابع فیزیکی زیرساخت‌ها می‌باشند. کوبرنتیز جهت پنهان کردن جزئیات از توسعه‌دهندگان این مفهوم را استفاده می‌کند. با استفاده از این مفهوم می‌توان تعاریف فیزیکی تعریف شده توسط Persistent Volume و Storage class را پنهان کرد.

۸. Ingress:

به صورت پیش فرض کوبرنتیز پادها را از محیط بیرون مستقل و ایزوله کرده است. اگر می‌خواهید کانالی برای ارتباط پادها با فضای بیرون ایجاد کنید از Ingress باید استفاده کنید.

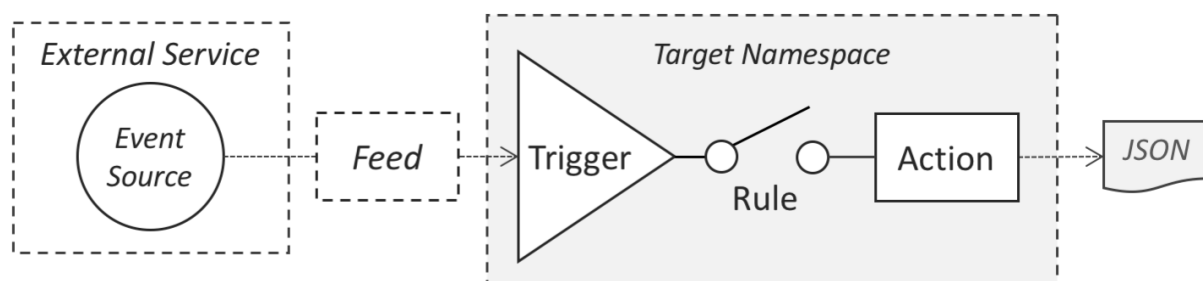
۵-۲ OpenWhisk

شرکت سرویس های وب آمازون با راه اندازی سرویس بدون سرور خود به اسم OpenWhisk در سال ۲۰۱۴ تغییر شگرفی در عرصه رایانش ابری ایجاد کرد. Lambda این امکان را به توسعه دهندگان نرم افزار داد که تنها با نوشتن تابع های خود به عنوان کنترل کننده های رویداد بتوانند هزاران رویداد را همزمان پردازش کنند. پس از معرفی Lambda گروهی در شرکت IBM متوجه ارزش بستر بدون سرور شدند و در سال ۲۰۱۵ تصمیم گرفتند که یک بستر بدون سرور را برای ابر IBM طراحی کنند. در ابتدا این بستر "Whisk" نام داشت که پس از گذشت یک سال متن باز شد و نام آن به OpenWhisk تغییر کرد. اکنون این پروژه قسمتی از Apache Software Foundation است و به عنوان اصلی ترین جایگزین متن باز برای Lambda حساب می شود. محصولات بدون سرور شرکت های IBM و Adobe با این بستر ساخته شده اند و همچنین بسیاری از شرکت های ارائه دهنده تلفن همراه و اینترنت از آن استفاده می کنند.

OpenWhisk یک بستر بدون سرور متن باز است که برای آسان کردن توسعه برنامه های کاربردی درون ابر طراحی شده است. برای معرفی و آشنایی بیشتر با این بستر ابتدا با توضیح معماری و ساختارش شروع می کنیم و سپس به بررسی نحوه عملکرد قسمت های مختلفش می پردازیم.

۱-۵-۲ معماری OpenWhisk

همانطور که در شکل ۱۷-۲ مشخص است، OpenWhisk تابع هایی به اسم action را در پاسخ به رویدادها اجرا می کند. این رویدادها می توانند توسط تایمرها، پایگاه داده ها، صف های پیام و یا وبسایت های مختلف تولید شده باشند. OpenWhisk کد منبع را با رابط خط فرمان (CLI) ورودی می گیرد و سرویس های خود را از راه شبکه اینترنت به مصرف کننده های مختلف مانند وب سایت ها، برنامه های کاربردی تلفن همراه و یا سرویس های REST API ارائه می دهد.



شکل ۱۷-۲: نحوه ی عملکرد رویداد محور OpenWhisk

۲-۵-۲ تابع‌ها و رویدادها

OpenWhisk با اجرا کردن تابع‌ها کارها را انجام می‌دهد. تابع تکه‌ای از کد است که چند ورودی دریافت و در پاسخ یک خروجی تولید می‌کند. یک تابع به طور کلی وضعیت را نگه نمی‌دارد (stateless)، اما برنامه‌های کاربردی که سمت سرور اجرا می‌شوند (backend) وضعیت را کاملاً حفظ می‌کنند (stateful). حفظ کردن وضعیت، مقیاس پذیری را محدود می‌کند زیرا در مقیاس‌های بالا به فضای خیلی زیادی برای نگهداری داده‌ها نیاز است. مهم‌تر از آن به سیستمی نیاز است که وضعیت بین فراخوانی‌های مختلف را همگام‌سازی کند. در نتیجه با افزایش بار سرور، زیرساخت عملیات حفظ و همگام‌سازی وضعیت تبدیل به مانعی در برابر رشد پروژه می‌شود. حفظ نکردن وضعیت اما این مزیت را دارد که می‌توان به راحتی چندین سرور دیگر را برای پروژه به کار گرفت بدون این که نیازی به همگام‌سازی وضعیتشان باشد. در OpenWhisk و به طور کلی در محیط بدون سرور تابع‌ها نباید وضعیت را نگهداری کنند. در محیط بدون سرور می‌توان وضعیت را نگهداری کرد اما نه در سطح یک تابع. برای نگهداری وضعیت باید از محل‌های ذخیره‌سازی مخصوصی استفاده شود که دارای قابلیت مقیاس پذیری بالا هستند.

OpenWhisk زیرساخت را مدیریت می‌کند و آن را برای رخ دادن اتفاق مهمی آماده نگه می‌دارد. این اتفاق مهم رویداد^{۴۹} نام دارد. تنها در صورتی یک تابع فراخوانی و اجرا می‌شود که رویدادی رخ داده باشد. این پردازش رویدادها در واقع مهم‌ترین عملیاتی است که محیط بدون سرور آن را مدیریت می‌کند. در نتیجه توسعه دهنده‌ها تنها باید برنامه‌ای بنویسند که به این رویدادها به طرز صحیحی پاسخ دهد و بقیه کارها به عهده ی ارائه دهنده ی سرویس می‌باشد.

۳-۵-۲ زبان‌های برنامه نویسی OpenWhisk

action های OpenWhisk را می‌توان با بسیاری از زبان‌های برنامه نویسی نوشت. اما معمولاً از زبان‌های تفسیر شده^{۵۰} استفاده می‌شود، مانند JavaScript، Python یا PHP. این زبان‌های برنامه نویسی به دلیل اجرا پذیر بودن بدون نیاز به کامپایل، در بازخورد طراحی خیلی سریع هستند. همچنین چون جزو زبان‌های سطح بالا محسوب می‌شوند، استفاده از آن‌ها نیز راحت‌تر است، اما در اجرا کندتر از زبان‌های کامپایل شده هستند.

علاوه بر زبان‌های تفسیر شده می‌توان از زبان‌های تفسیر شده‌ی از قبل کامپایل شده مانند Java و Scala استفاده کرد. این زبان‌ها توسط ماشین مجازی Java یا به اختصار JVM اجرا می‌شوند.

در نهایت از زبان‌های کامپایل شده نیز در OpenWhisk می‌توان استفاده کرد. در این حالت یک

⁴⁹event

⁵⁰interpreted

فایل باینری اجراپذیر بر روی سیستم بدون هیچ واسطه‌ای اجرا می‌شود. OpenWhisk تنها از زبان های Go و Swift در این دسته از زبان ها پشتیبانی می‌کند.

به جز زبان های گفته شده از هر نوع زبان و سیستمی نیز برای تعریف action های OpenWhisk می‌توان استفاده کرد به این شرط که به عنوان یک ایمج داکر بسته بندی شود و در Docker hub منتشر شود. OpenWhisk می‌تواند با دریافت ایمج، آن را اجرا کند.

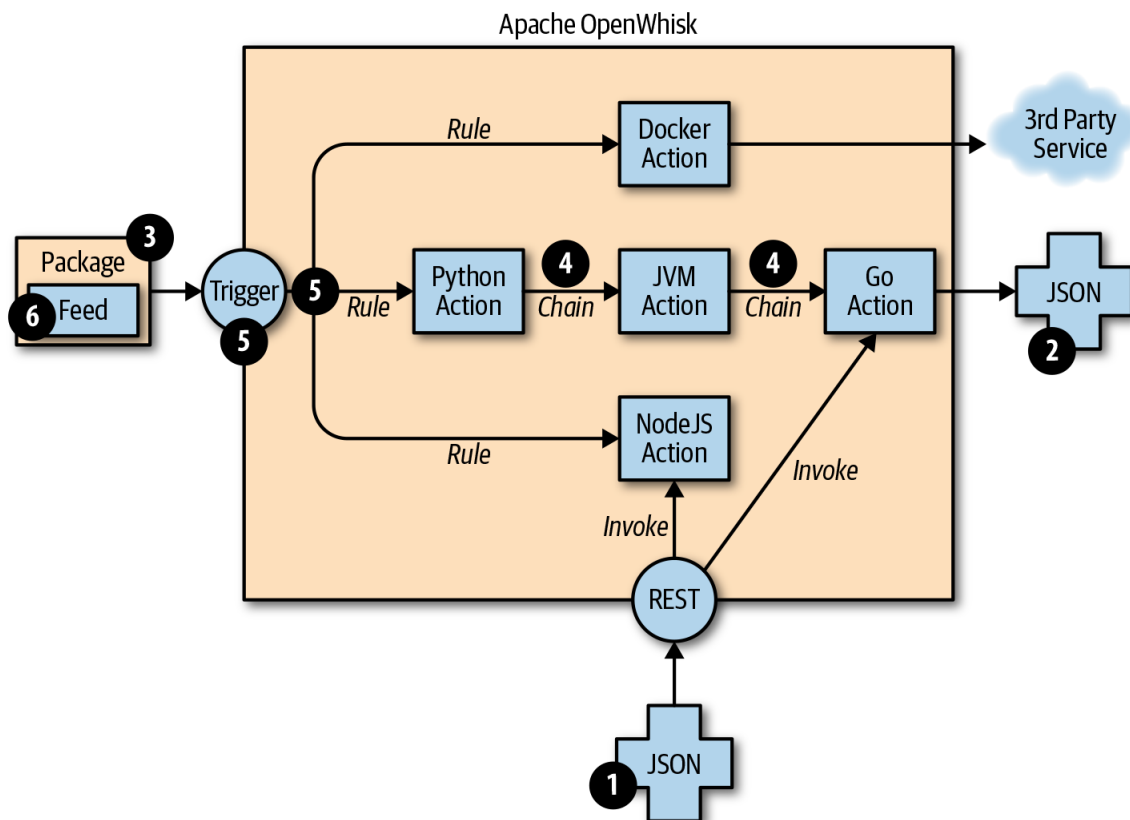
۴-۵-۲ Action ها

برنامه های ساخته شده با OpenWhisk مجموعه ای متشکل از action ها هستند. این action ها تکه ای کد هستند که با زبان های گفته شده نوشته شده اند و می‌توان آن ها را فراخوانی و اجرا کرد (invoke). در زمان فراخوانی action، تعدادی ورودی با فرمت JSON به تابع آن داده می‌شود. سپس در انتهای اجرا، action باید یک خروجی تولید کند که آن هم نیز باید با فرمت JSON بازگشت داده شود. action ها را می‌توان با استفاده از package ها دسته بندی نمود. یک package را می‌توان به وسیله binding ها با بقیه user ها به اشتراک گذاشت. همچنین برای یک package می‌توان مؤلفه هایی تعیین کرد که مختص هر binding باشند و به action هایش به ارث برسند.

۵-۵-۲ زنجیر کردن Action ها

action ها از راه های مختلفی می‌توانند به یکدیگر متصل شوند. ساده ترین راه متصل کردن به وسیله sequence است. action های متصل به هم از خروجی action قبل از خود به عنوان ورودی استفاده می‌کنند.

بسیاری از جریان های مورد نیاز برای پیاده سازی منطق برنامه را نمی‌توان به صورت یک مسیر خطی تعریف کرد که تنها یک ورودی می‌گیرند و در انتها یک خروجی را برمی گردانند. بنابراین راهی وجود دارد که می‌توان جریان action ها را به مسیرهای مختلف تقسیم کرد. این ویژگی با trigger ها (ماشه ها) و rule ها (قوانین) پیاده سازی می‌شود. یک trigger به تنهایی کاری انجام نمی‌دهد، اگرچه می‌توان آن را با یک یا چند action از طریق تعریف کردن rule مرتبط کرد. پس از تعریف کردن trigger و مرتبط کردن آن با چندین action می‌توان آن را با دادن مؤلفه ها شلیک (fire) کرد. در شکل ۱۸-۲ معماری action های OpenWhisk نشان داده شده است.



شکل ۲-۱۸: نمای کلی ساختار action های OpenWhisk

برای شلیک کردن trigger یک action باید تعریف کرد که feed نام دارد. feed باید طبق الگوی طراحی ناظر^{۵۱} طراحی شود و بتواند یک trigger را در هنگام رخ دادن یک رویداد فعال کند. الگوی ناظر یک الگوی طراحی نرم افزار است که در آن یک شی به نام موضوع، فهرست وابستگی هایش را با نام ناظران نگه می دارد و هرگونه تغییر در وضعیتش را به طور خودکار و معمولاً با صدا کردن یکی از روش های آن به اطلاع آن اشیا می رساند. پس از طراحی feed در هنگام پیاده سازی برنامه می توان آن را با یک trigger ادغام نمود تا آن trigger نیز در هنگام شلیک، چند action دیگر را فراخوانی کند.

⁵¹observer

۶-۵-۲ نحوه عملکرد OpenWhisk

حال پس از آشنایی با اجزاء مختلف OpenWhisk به بررسی نحوه عملکرد آن‌ها می‌پردازیم. OpenWhisk با بهره‌گیری از چندین پروژه معروف و توسعه یافته متن باز ساخته شده است که در زیر معرفی شده‌اند: (شکل ۱۹-۲)

Nginx: یک وب سرور با کارایی بالا و پروکسی معکوس

CouchDB: پایگاه داده سند محور، NoSQL و مقیاس پذیر

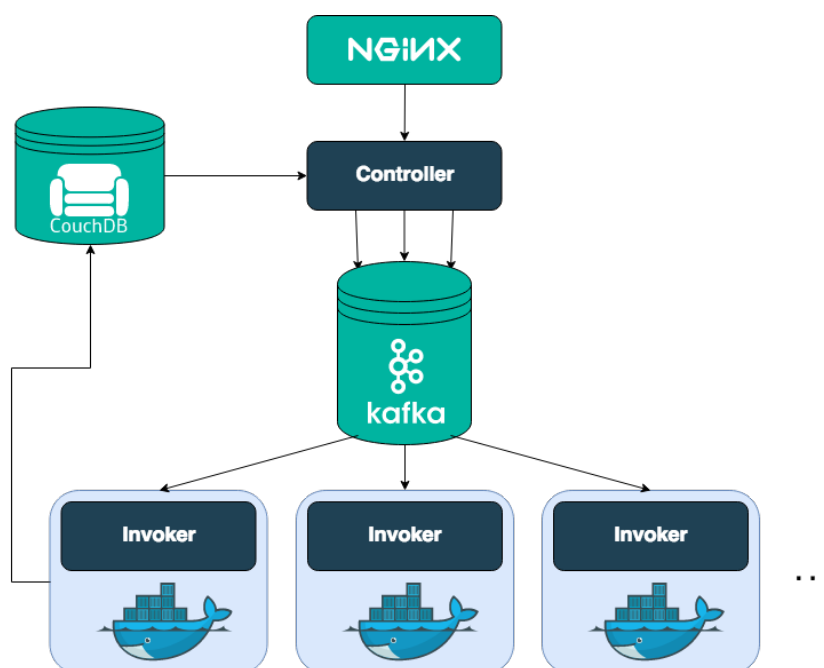
Kafka: سیستم پیام رسانی انتشار-اشتراک توزیع شده با کارایی بالا و مقیاس پذیر

تمامی این اجزاء کانتینرهای داکر هستند و می‌توانند توسط محیط‌هایی که از این فرمت پشتیبانی می‌کنند مانند کورنتیز اجرا شوند.

OpenWhisk شامل قسمت‌های دیگری می‌باشد که توسط تیم خودش ساخته شده است: **Controller**: موجودیت‌های مختلف را مدیریت می‌کند، triggerها را شلیک می‌کند و همچنین فراخوانی actionها را مسیریابی می‌کند.

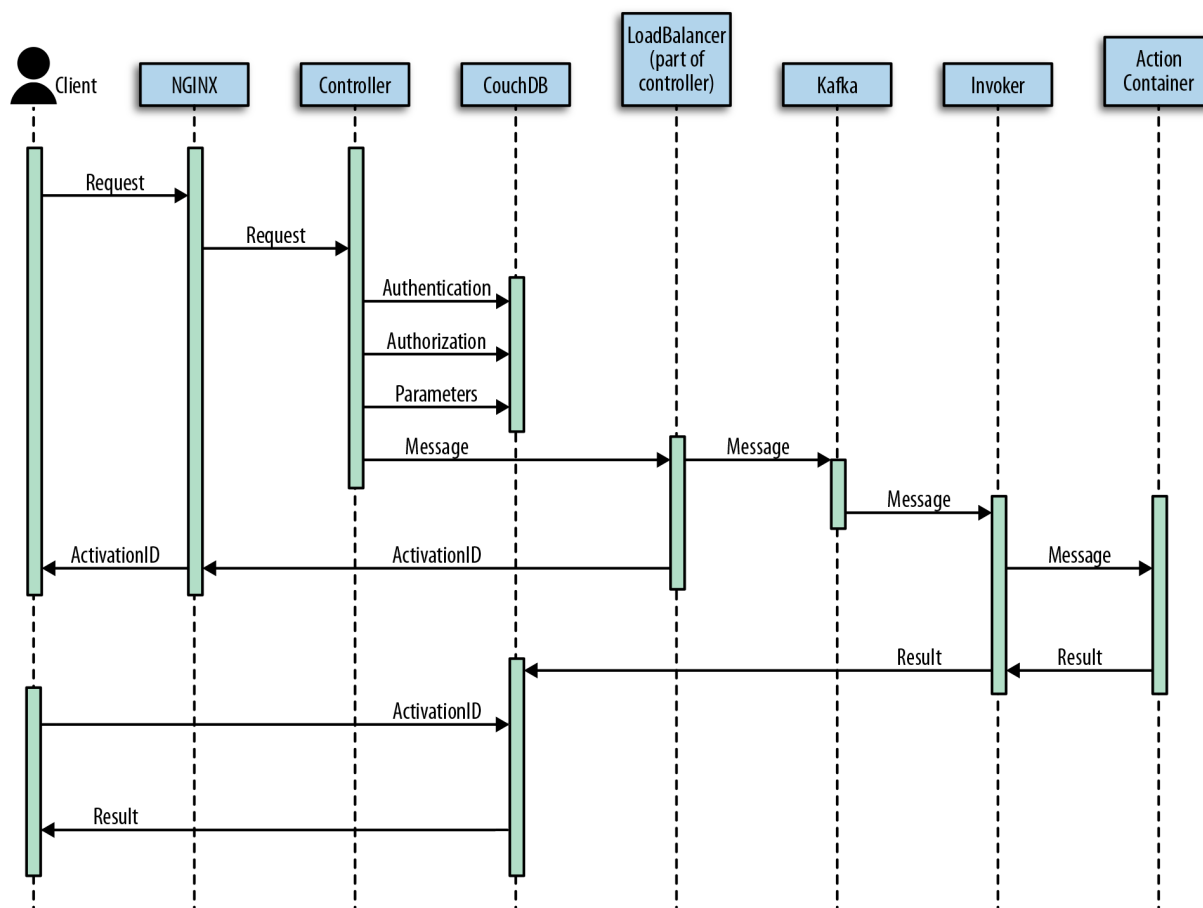
Invoker: کانتینرها را برای اجرای actionها راه اندازی می‌کند.

Action containers: کانتینرهای حاوی action که در واقع actionها را اجرا می‌کنند.



شکل ۱۹-۲: ساختار OpenWhisk

تمامی پردازشی که در OpenWhisk انجام می‌شود ناهمگام^{۵۲} است. این پردازش شامل دریافت، کنترل، صف‌بندی، پردازش و ذخیره‌سازی نتایج یک درخواست فراخوانی action است. حال با جزئیات بیشتری به بررسی این مراحل می‌پردازیم. نمودار زمانی این مراحل در شکل ۲-۲۰ نشان داده شده است.



شکل ۲-۲۰: چگونگی پردازش action در OpenWhisk

Nginx

همه چیز با فراخوانی action آغاز می‌شود. این فراخوانی از راه های مختلفی امکان پذیر است:

- از طریق وب هنگامی که action به عنوان web action تعریف شده باشد.
- فراخوانی از action دیگر با استفاده از API
- هنگامی که یک trigger فعال شود و یک rule برای فراخوانی action وجود داشته باشد.
- از طریق CLI

⁵²asynchronous

OpenWhisk یک سیستم RESTful است، بنابراین هر فراخوانی action به یک درخواست HTTPS تبدیل می‌شود و به نودی لبه که همان Nginx است می‌رسد. دلیل اصلی وجود وب سرور Nginx، پیاده‌سازی و پشتیبانی از پروتکل امن HTTPS می‌باشد. Nginx پس از دریافت درخواست آن را به سرویس اصلی میانی به نام Controller می‌فرستد.

Controller

قبل از اجرا شدن action، ابتدا controller امکان اجرای و صحت آن را بررسی می‌کند. سپس منبع آن هویت سنجی می‌شود تا اجازه دسترسی آن مشخص شود. اگر action از دو مرحله قبل عبور کرد، مؤلفه‌های اضافی به عنوان پیکربندی به آن افزوده می‌شود. حال که امکان اجرای action تأیید شد، به قسمت بعدی که Load Balancer است فرستاده می‌شود.

Load Balancer

وظیفه‌ی Load Balancer همان طور که از نامش پیداست حفظ تعادل میان اجرا کننده‌های action یا همان invoker های OpenWhisk است. Load Balancer با مواظبت از runtime های action موجود، اگر دوباره مورد نیاز باشد از آن استفاده می‌کند و اگر runtime مورد نیاز موجود نبود آن را می‌سازد.

Kafka

به جایی رسیدیم که سیستم آماده اجرای action است. با این حال، نمی‌توان آن action را فوراً به یک invoker ارسال کرد، زیرا ممکن است مشغول اجرای یک action دیگر باشد. همچنین این احتمال وجود دارد که یک invoker خراب شود، یا حتی کل سیستم خراب شده و دوباره راه اندازی شود. بنابراین، از آنجا که ما در یک محیط کاملاً موازی کار می‌کنیم که انتظار می‌رود مقیاس پذیر باشد، باید این احتمال را در نظر بگیریم که منابع مورد نیاز خود را برای اجرای فوری action در دسترس نداشته باشیم. در مواردی از این دست، مجبوریم فراخوانی‌ها را بافر کنیم. OpenWhisk از Kafka برای انجام این عمل استفاده می‌کند. Kafka یک سیستم پیام رسان "انتشار - اشتراک" با عملکرد بالا است که می‌تواند درخواست‌ها را تا زمان آماده شدن برای اجرای آن‌ها ذخیره کند. درخواست که برای ورود به Nginx به HTTPS تبدیل شده بود، توسط Load Balancer به یک پیام Kafka تبدیل می‌شود که به invoker مقصدش آدرس دهی شده است.

هر پیام ارسال شده به یک invoker شناسه‌ای دارد به نام activation-ID. هنگامی که پیام در صف Kafka قرار می‌گیرد، دو امکان وجود دارد: فراخوانی بدون انسداد و مسدود کننده. برای یک فراخوانی بدون انسداد، شناسه فعال سازی به عنوان پاسخ نهایی درخواست به client ارسال می‌شود و درخواست

تکمیل می‌شود. در این حالت، پیش‌بینی می‌شود client بعداً برگردد تا نتیجه فراخوانی را بررسی کند. برای یک فراخوانی مسدود کننده، اتصال باز می‌ماند؛ کنترل کننده منتظر نتیجه action است و نتیجه را برای سرویس‌گیرنده ارسال می‌کند.

Invoker

در OpenWhisk، بخش invoker وظیفه اجرای action را بر عهده دارد. action ها در واقع در محیط های جدا شده که توسط کانتینرهای Docker ساخته شده اند، اجرا می‌شوند به این صورت که invoker ابتدا ایمج runtime مورد نیاز برای اجرای action را انتخاب می‌کند و سپس آن را همراه با کد action راه اندازی می‌کند.

پس از اینکه runtime اجرا شد، action هایی که تا آن زمان ساخته و آماده شده اند توسط invoker به runtime فرستاده می‌شوند. invoker همچنین log هایی که مربوط به اجرای action ها هستند را مدیریت و ذخیره می‌کند.

CouchDB

پس از اتمام پردازش، OpenWhisk نتیجه را در پایگاه داده CouchDB ذخیره می‌کند. سپس تمامی نتایج اجرای action ها که در پایگاه داده ذخیره شده‌اند توسط activation-ID که به client ارسال شده بود، در دسترس هستند.

Client

پردازشی که اکنون توضیح داده شد ناهمگام بود. بدان معنی که سرویس‌گیرنده درخواستی را آغاز می‌کند و سپس آن را کنار می‌گذارد، اگرچه آن را به طور کامل فراموش نمی‌کند، زیرا یک شناسه فعال‌سازی را به عنوان نتیجه‌ی فراخوانی دریافت کرده است. از activation-ID برای ذخیره نتیجه در پایگاه داده پس از پردازش استفاده می‌شود. برای بازیابی نتیجه نهایی، سرویس‌گیرنده باید درخواست دیگری را همراه با شناسه فعال‌سازی به عنوان مؤلفه ورودی ارسال کند. پس از اتمام action، نتیجه، log ها و سایر اطلاعات در پایگاه داده موجود است و قابل بازیابی است. پردازش همگام هم امکان پذیر است که مانند همان روش پردازش ناهمگام کار می‌کند، اما مسدود کننده است به این معنی که سرویس‌گیرنده منتظر اجرای کامل action می‌ماند و نتیجه را فوراً دریافت می‌کند [۱۴].

فصل سوم

طراحی و پیاده سازی

در این بخش ابتدا پروژه به طور کلی شرح داده شده و در ادامه طراحی بستر بدون سرور اینترنت اشیاء به صورت میکروسرویس توضیح داده می شود. پس از معرفی ابزارها و تکنولوژی های استفاده شده، نحوه پیاده سازی این بستر مورد بررسی قرار می گیرد.

۱-۳ دید کلی

بستر های اینترنت اشیاء بسته به کاربرد، نوع دستگاه ها و محیط مورد استفاده با قابلیت های مختلفی طراحی می شوند. به عنوان مثال یک بستر خانه هوشمند با یک بستر حمل و نقل هوشمند می تواند تفاوت های زیادی در معماری، نحوه ذخیره سازی، مدیریت و نمایش داده ها داشته باشد اما در بسیاری از موارد یکسان هستند. برخی از این موارد عبارتند از:

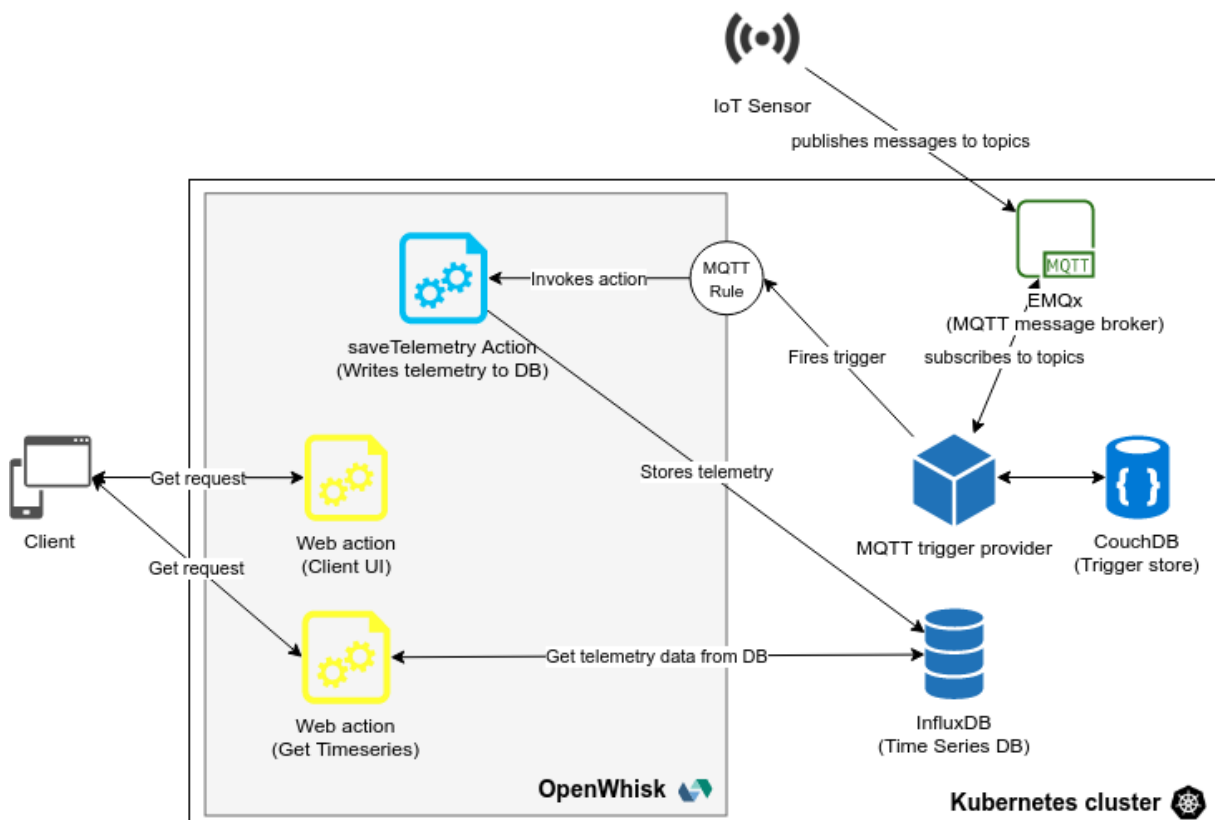
- جمع آوری و ذخیره سازی داده های حسگرها به صورت سری زمانی
- کنترل دستگاه ها و ارسال فرمان به آنها
- مدیریت دستگاه ها و کاربران
- برخورد مناسب با رویدادها
- پردازش و تحلیل داده ها

مهم ترین قابلیت یک بستر اینترنت اشیاء جمع آوری و ذخیره سازی داده های حسگرها می باشد. این داده ها معمولاً به صورت یک پیام با پروتکل MQTT به یک message broker ارسال می شود، پس از احراز هویت دستگاه، به هسته بستر ارسال شده و برای پردازش های لازم مورد استفاده قرار می گیرد. این پیام همچنین به پایگاه داده فرستاده شده و به صورت سری زمانی ذخیره می گردد. سپس این داده ها می توانند توسط بخش های دیگر بستر استخراج و برای تحلیل و مصورسازی به سرویس گیرنده ارسال شود. در این پروژه این قابلیت با استفاده از بستر بدون سرور OpenWhisk و هماهنگ کننده ابری کوبرنتیز طراحی و پیاده سازی شده است.

۲-۳ طراحی

طراحی این پروژه از چهار بخش اصلی تشکیل شده است. اولین بخش طراحی معماری میکروسرویس message broker برای دریافت و ارسال داده‌های حسگر، پایگاه داده برای ذخیره سازی داده‌های سری زمانی و یک پایگاه داده دیگر برای ذخیره سازی trigger است. بخش دوم طراحی یک مازول سرویس‌دهنده به اسم MQTT trigger provider است که پیام را از message broker دریافت کرده آن را از طریق سیستم trigger و rule به بستر بدون سرور ارسال می‌کند. بخش سوم شامل طراحی action در بستر بدون سرور OpenWhisk برای ذخیره سازی و استخراج داده‌های سری زمانی است. در بخش چهارم طراحی یک برنامه کاربردی با استفاده از این بستر در دو بخش سرویس‌دهنده برای ارائه‌ی سرویس‌های دلخواه و بخش سرویس‌گیرنده برای نمایش داده‌ها بررسی می‌شود.

نمای کلی معماری پروژه در شکل ۱-۳ نشان داده شده است.



شکل ۱-۳: نمای کلی معماری پروژه

۱-۲-۳ طراحی میکروسرویس

این بستر به صورت میکروسرویس طراحی شده است و عملکردهای مختلف این بستر توسط این میکروسرویس‌ها انجام می‌شوند. در ادامه طراحی معماری این میکروسرویس‌ها را بررسی می‌کنیم:

- **message broker**: این سرویس برای اتصال دستگاه‌های اینترنت اشیاء به بستر طراحی شده است و پیام‌های ارسالی از طرف دستگاه‌ها را باید به سرویس‌های دیگر بستر که مشترک شده‌اند بفرستد. بنابراین نیاز است که این سرویس از بیرون و درون بستر قابل دسترسی باشد. برای دسترسی از بیرون از قابلیت expose کردن port توسط کورنتیز استفاده شده است. همچنین یک سرویس کورنتیز برای اتصال به دیگر سرویس‌های درونی بستر طراحی شده است.
- **trigger store**: این سرویس یک پایگاه داده CouchDB است که باید trigger ها را ذخیره کند. دسترسی و اتصال به این سرویس از داخل بستر پس از راه‌اندازی به راحتی امکان پذیر است و تنها نیاز به تنظیم کردن و مهیا نمودن پایگاه داده برای ذخیره و بازیابی داده‌های مورد نیاز است.
- **timeseries database**: پایگاه داده سری زمانی InfluxDB است. این پایگاه داده برای ذخیره‌سازی داده‌های سری زمانی دستگاه‌ها طراحی شده است و پس از راه‌اندازی توسط سرویس‌های دیگر درون بستر در دسترس است.
- **OpenWhisk**: پیاده‌سازی و راه‌اندازی بستر OpenWhisk بر روی بستر کورنتیز توسط یک پروژه که به همین منظور طراحی شده است، وجود دارد. این پروژه، اجزای OpenWhisk را به صورت اشیاء کورنتیز درون یک namespace جدا راه‌اندازی می‌کند. سرویس‌هایی طراحی شده است که ارتباط این بستر را با سرویس‌های دیگر درون namespace اصلی برقرار می‌سازد.

۲-۲-۳ MQTT trigger provider

این قسمت از بستر برای اشتراک و دریافت پیام از message broker و ارسال آن به بستر بدون سرور برای ذخیره سازی طراحی شده است. با افزوده شدن یک trigger در بستر بدون سرور برای دریافت پیام ها بر روی یک topic جدید، این سرویس آن را دریافت می‌کند و بر روی آن topic مشترک می‌شود. همچنین trigger ها را در یک پایگاه داده ذخیره می‌کند تا در هنگام راه‌اندازی مجدد سرویس بتواند دوباره بر روی آن topic ها مشترک شود.

۳-۲-۳ مدیریت داده‌های سری زمانی

پس از شلیک شدن trigger توسط سرویس MQTT trigger provider، پیام باید پردازش و داده موجود در آن در پایگاه داده ذخیره شود. یک action با نام saveTelemetry برای این کار طراحی شده است و trigger شلیک شده با تعریف کردن یک rule به آن متصل می‌شود. این action یک تابع است که با دریافت پیام آن را باز کرده و با فرمت مشخص برای ذخیره سازی به پایگاه داده سری زمانی می‌فرستد. برای استخراج داده‌های سری زمانی از پایگاه داده یک action دیگر به اسم getTimeSeries طراحی شده است که با گرفتن token دستگاه و کلیدهای داده‌ها، داده‌های مشخص شده با کلید و مربوط به آن دستگاه را به صورت سری زمانی برمی‌گرداند. این action همچنین پارامتر دیگری به نام limit برای محدود کردن تعداد داده‌های سری زمانی دارد و اگر برابر n باشد به این معنی است که n تا از آخرین داده‌ها در پاسخ ارسال شوند. در صورتی که این پارامتر وجود نداشته باشد تنها آخرین داده ارسال خواهد شد. برای این action دسترسی وب نیز تعریف می‌شود تا بتوان آن را از طریق درخواست HTTP اجرا کرد.

۴-۲-۳ برنامه کاربردی

تا اینجا طراحی بستر به پایان رسید. پس از ایجاد trigger با topic مشخص، دستگاه می‌تواند داده‌های خود را در قالب پیام‌های MQTT به broker ارسال کند و این داده‌ها در پایگاه داده ذخیره شده و در دسترس خواهند بود. حال کاربران می‌توانند به توسعه برنامه کاربردی خود بپردازند. با تعریف تابع‌ها در قالب action می‌توان از داده‌های سری زمانی ذخیره شده برای پردازش‌های مختلف استفاده کرد. با تعریف کردن rule می‌توان سناریوهای دلخواهی تعریف کرد که در هنگام دریافت پیام یا شلیک trigger تعریف شده اجرا شوند. همچنین معماری RESTful APIs نیز با تعریف کردن action ها به عنوان web action به سادگی امکان پذیر است.

حال برای نمونه یک برنامه‌ی کاربردی با استفاده از action ها طراحی شده است. این برنامه در ابتدا دستگاه‌های موجود را گرفته و نمایش می‌دهد و پس از انتخاب هر دستگاه، داده‌های ارسالی توسط آن دستگاه به صورت بلادرنگ نمایش داده می‌شود.

۳-۳ تکنولوژی‌های استفاده شده

۱-۳-۳ MicroK8s

MicroK8s این امکان را فراهم می‌سازد تا با اجرای یک دستور، یک نود کوبرنتیز برای ایجاد یک محیط توسعه و تست به صورت محلی نصب و اجرا شود. نصب و اجرای MicroK8s بسیار سریع و آسان است و بسیاری از بسته‌های مورد نیاز برای توسعه‌ی محلی برنامه‌های کاربردی را شامل می‌شود. از آن جایی که راه‌اندازی کوبرنتیز کمی تخصصی و پیچیده است و همچنین بسته‌های مورد نیاز دیگری باید به صورت دستی بر روی آن نصب شود، استفاده از یک ابزار که تمامی این کارها را به صورت خودکار انجام می‌دهد بسیار مفید است [۷].

ابزار دیگری به نام MiniKube وجود دارد که یک نود کوبرنتیز را بر روی یک ماشین مجازی درون سیستم عامل راه‌اندازی می‌کند. این ابزار نیز مخصوص توسعه و تست است. این ابزار بر روی تمامی سیستم عامل‌ها قابل اجرا است، اما MicroK8s فقط از سیستم عامل لینوکس^۱ پشتیبانی می‌کند [۸].

۲-۳-۳ Helm

ابزار مدیریت پکیج‌هایی است که برای پیاده‌سازی بر روی بستر کوبرنتیز تنظیم شده‌اند. این منابع از پیش تنظیم شده با نام charts شناخته می‌شوند. با کمک Helm می‌توان پکیج‌ها و منابعی که به صورت شخصی ساخته می‌شود را به اشتراک گذاشت. همچنین امکان به‌روز رسانی محصولات توسط این ابزار وجود دارد [۴].

۳-۳-۳ EMQx

EMQx یک message broker توزیع شده با پروتکل MQTT برای اینترنت اشیاء، ارتباط ماشین به ماشین^۲ و برنامه‌های کاربردی تلفن همراه است؛ کاملاً متن باز، بسیار مقیاس‌پذیر و بسیار در دسترس است که یک خوشه آن می‌تواند ده‌ها میلیون سرویس‌گیرنده را همزمان اداره کند. بیش از ۵ هزار شرکت از این message boker برای اتصال به ۵۰ میلیون دستگاه اینترنت اشیاء استفاده می‌کنند [۳].

۴-۳-۳ CouchDB

CouchDB یک پایگاه داده NoSQL، سند محور و متن باز است که با زبان همگام Erlang پیاده‌سازی شده است. از JSON برای ذخیره داده‌ها، از زبان JavaScript به عنوان زبان اجرای دستور و از HTTP

^۱Linux

^۲M2M

برای API استفاده می کند.

برخلاف یک پایگاه داده رابطه‌ای^۳، پایگاه داده CouchDB داده‌ها و روابط را در جدول‌ها ذخیره نمی کند. در عوض، هر بانک اطلاعاتی مجموعه ای از اسناد مستقل است. ویژگی متمایز کننده‌ی CouchDB همانندسازی مولتی-مستر^۴ است که به آن اجازه می دهد تا در دستگاه‌ها مقیاس شود و سیستم‌های با کارایی بالا بسازد [۲].

InfluxDB ۵-۳-۳

InfluxDB یک پایگاه داده سری زمانی^۵ متن باز است که توسط شرکت InfluxData و با زبان Go نوشته شده است. همچنین برای ذخیره سازی سریع، در دسترس بودن بالا و بازیابی داده‌های سری زمانی در زمینه هایی مانند نظارت بر عملیات، اندازه گیری برنامه‌ها، داده‌های حسگرهای اینترنت اشیا و تجزیه و تحلیل بلادرنگ بهینه شده است. این پایگاه داده از زبان شبیه به SQL برای اجرای دستورات استفاده می کند [۵].

³relational database

⁴multi-master replication

⁵Time Series Data Base (TSDB)

۴-۳ پیاده سازی

حال پس از بررسی طراحی و آشنایی با تکنولوژی‌های مورد نیاز به پیاده سازی این بستر می‌پردازیم. اولین مرحله‌ی پیاده سازی، فراهم نمودن یک خوشه از هماهنگ کننده ابری کوبرنتیز است. این هماهنگ کننده می‌تواند از سرویس ارائه دهندگان ابری تهیه شده یا به صورت دستی روی سرورهای خصوصی پیاده سازی شود. برای پیاده سازی محلی نیز می‌توان از برنامه‌ی minikube استفاده کرد. این برنامه یک ماشین مجازی به صورت محلی می‌سازد و کوبرنتیز را درون آن ماشین مجازی پیاده سازی و اجرا می‌کند. در این پروژه از MicroK8s برای این منظور استفاده شده است.

پس از اجرای هماهنگ کننده ابری نوبت به پیاده‌سازی سایر میکروسرویس‌ها می‌رسد. برای راه اندازی میکروسرویس‌ها ابتدا باید تنظیم کننده‌ی بسته Helm نصب و راه اندازی شود. سپس با استفاده از Helm بسته‌های EMQx، پایگاه داده CouchDB و پایگاه داده سری زمانی InfluxDB از طریق chart هایشان نصب و اجرا می‌شوند. هر میکروسرویس پس از اجرا باید تنظیمات مورد نیازش انجام شود و برای اطمینان از صحت عملکرد مورد تست قرار گیرد.

۱-۴-۳ OpenWhisk

پیاده سازی OpenWhisk نیز توسط Helm انجام می‌شود. ابتدا باید پروژه‌ای که به همین منظور ساخته شده است را متناسب با نحوه‌ی پیاده‌سازی تنظیم کرده و مورد استفاده قرار داد. این تنظیمات عبارتند از فعال سازی یک نود ingress و تعریف کردن host name و port برای ایجاد راه دسترسی به OpenWhisk. این نود با اتصال به NginX درون OpenWhisk ارتباط با این بستر را از طریق وب میسر می‌سازد. همچنین تنظیمات دیگری برای نحوه‌ی ذخیره سازی داده‌های پایگاه داده CouchDB توسط کوبرنتیز امکان‌پذیر است. در انتها این تنظیمات درون یک فایل ذخیره شده و OpenWhisk با اجرای یک دستور CLI در یک namespace جدا راه اندازی می‌شود.

۲-۴-۳ MQTT trigger provider

این مازول برای راه اندازی در بستر کوبرنتیز باید به صورت کانتینر در بیاید. بنابراین یک فایل داکر برای ساخت کانتینر نوشته شده است. ایمج این کانتینر توسط این فایل ساخته شده و در Docker Hub آپلود می‌شود. پس از آن فایلی برای تعریف یک deployment نوشته می‌شود تا این کانتینر را در بستر کوبرنتیز راه اندازی کند. در این فایل پس از مشخص کردن اطلاعات ایمج کانتینر و deployment باید port کانتینر و متغیرهای محیطی لازم برای اتصال به سایر سرویس‌ها مشخص شود. این متغیرها عبارتند

از نام کاربری، رمزعبور، آدرس و port مورد نیاز برای اتصال به پایگاه داده CouchDB و همچنین آدرس سرویس OpenWhisk. اطلاعات رمزعبور و نام کاربری از طریق secret کوبرنتیز در دسترس هستند. فایل تنظیمات deployment در شکل ۲-۳ نشان داده شده است.

```

< > mqtt-trigger-provider-deployment.yml x
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: mqtt-trigger-provider
5  spec:
6    replicas: 1
7    selector:
8      matchLabels:
9        app: mqtt-trigger-provider
10   template:
11     metadata:
12       labels:
13         app: mqtt-trigger-provider
14     spec:
15       containers:
16       - name: mqtt-trigger-provider
17         image: docker.io/hassanmojab/mqtt-trigger-provider:latest
18         imagePullPolicy: IfNotPresent
19         ports:
20         - containerPort: 3000
21         env:
22         - name: COUCHDB_USER
23           valueFrom:
24             secretKeyRef:
25               name: triggersdb-couchdb
26               key: adminUsername
27         - name: COUCHDB_PASS
28           valueFrom:
29             secretKeyRef:
30               name: triggersdb-couchdb
31               key: adminPassword
32         - name: COUCHDB_HOST
33           value: triggersdb-couchdb
34         - name: COUCHDB_HOST_PORT
35           value: '5984'
36         - name: OPENWHISK_API_HOSTNAME
37           value: openwhisk-host

```

شکل ۲-۳: سرویس‌های بستر بدون سرور اینترنت اشیاء

۳-۴-۳ سرویس‌های کوبرنتیز

برای متصل شدن بقیه اجزاء با یکدیگر باید سرویس‌های دیگری تعریف شود. این سرویس‌ها در قالب فایل‌های مجزا تعریف شده و در انتها با اجرای یک دستور CLI توسط کوبرنتیز خوانده و اعمال می‌شوند. این سرویس‌ها عبارتند از:

- سرویس cluster IP برای اتصال EMQx و OpenWhisk به MQTT trigger provider
- سرویس cluster IP برای اتصال OpenWhisk NginX
- سرویس external name برای اتصال به سرویس OpenWhisk از namespace اصلی
- سرویس external name برای اتصال به سرویس MQTT trigger provider از namespace مربوط به OpenWhisk
- سرویس external name برای اتصال به سرویس پایگاه داده InfluxDB از namespace مربوط به OpenWhisk

پس از اعمال، سرویس‌های در حال اجرا توسط دستور CLI قابل مشاهده هستند. سرویس‌های در حال اجرای این بستر در namespace های default و openwhisk در شکل ۳-۳ نشان داده شده است.

```
[hassan@tok-server:~]$ kubectl get svc
NAME                                TYPE                CLUSTER-IP          EXTERNAL-IP
emqx                               ClusterIP           10.152.183.75       <none>
emqx-headless                      ClusterIP           None                 <none>
emqx-mqtt                          NodePort            10.152.183.211      <none>
kubernetes                         ClusterIP           10.152.183.1        <none>
mqtt-trigger-provider              ClusterIP           10.152.183.137      <none>
openwhisk-host                    ExternalName         <none>               openwhisk-nginx.openwhisk.svc.cluster.local
triggersdb-couchdb                ClusterIP           None                 <none>
triggersdb-couchdb-node-port      NodePort            10.152.183.60       <none>
ts-influxdb                       ClusterIP           10.152.183.15       <none>
[hassan@tok-server:~]$ kubectl get svc -n openwhisk
NAME                                TYPE                CLUSTER-IP          EXTERNAL-IP
mqtt-trigger-provider              ExternalName         <none>               mqtt-trigger-provider.default.svc.cluster.local
openwhisk-nginx                   ClusterIP           10.152.183.70       <none>
owdev-apigateway                  ClusterIP           10.152.183.204      <none>
owdev-controller                  ClusterIP           10.152.183.114      <none>
owdev-couchdb                    ClusterIP           10.152.183.205      <none>
owdev-kafka                       ClusterIP           None                 <none>
owdev-nginx                      NodePort            10.152.183.157      <none>
owdev-redis                      ClusterIP           10.152.183.30       <none>
owdev-zookeeper                  ClusterIP           None                 <none>
triggersdb-couchdb-host           ExternalName         <none>               triggersdb-couchdb.default.svc.cluster.local
ts-influxdb-host                  ExternalName         <none>               ts-influxdb.default.svc.cluster.local
```

شکل ۳-۳: سرویس‌های بستر بدون سرور اینترنت اشیاء

۴-۴-۳ action ها و trigger ها

برای پیاده سازی action ها و trigger ها در OpenWhisk از رابط CLI آن استفاده می‌شود. ابتدا باید action feed طراحی شده برای ایجاد و حذف trigger ها در MQTT trigger provider راه اندازی شود. بنابراین توسط کاربر ادمین OpenWhisk یک package به نام mqtt ساخته شده و برای همه‌ی کاربران به اشتراک گذاشته می‌شود. سپس تابع feed action درون این package آپلود می‌شود. حال کاربر می‌تواند یک trigger با feed تعریف شده برای یک topic مشخص، راه اندازی کند و پس از آن تابع saveTelemetry را به عنوان action آپلود کند. در انتها با تعریف کردن یک rule اتصال trigger به این action برقرار می‌شود. اجرای دستورات تمامی این مراحل در شکل ۴-۳ قابل مشاهده است.

```
(base) Hassans-MacBook-Pro:tok hassanmojab$ wsk -i package create mqtt --shared yes \
> -p providerEndpoint 'http://mqtt-trigger-provider:3000/mqtt'
ok: created package mqtt
(base) Hassans-MacBook-Pro:tok hassanmojab$ wsk -i action create \
> -a feed true mqtt/mqttFeed actions/feedAction.zip \
> --kind nodejs:default
ok: created action mqtt/mqttFeed
(base) Hassans-MacBook-Pro:tok hassanmojab$ wsk -i trigger create feedTrigger \
> --feed /whisk.system/mqtt/mqttFeed \
> -p topic 'topic' -p url 'mqtt://emqx' &>/dev/null
(base) Hassans-MacBook-Pro:tok hassanmojab$ wsk -i action create saveTelemetry \
> --kind nodejs:default actions/saveTelemetry.zip
ok: created action saveTelemetry
(base) Hassans-MacBook-Pro:tok hassanmojab$ wsk -i rule create mqttRule /_/feedTrigger saveTelemetry
ok: created rule mqttRule
```

شکل ۴-۳: ساخت action feed، trigger و rule

تابع طراحی شده برای دریافت داده‌های سری زمانی نیز به عنوان یک web action آپلود شده و توسط یک دستور CLI می‌توان آدرس آن را در قالب URL دریافت کرد (شکل ۵-۳).

```
((base) Hassans-MacBook-Pro:tok hassanmojab$ wsk -i action create clientUI actions/client/clientUI.js --web true
ok: created action clientUI
(base) Hassans-MacBook-Pro:tok hassanmojab$ wsk -i action create getTimeseries actions/api/getTimeseries.zip --kind
nodejs:default --web true
ok: created action getTimeseries
(base) Hassans-MacBook-Pro:tok hassanmojab$ wsk -i action get clientUI --url
ok: got action clientUI
https://172.160.220.160:31001/api/v1/web/guest/default/clientUI
```

شکل ۵-۳: آپلود web action و دریافت آدرس آن

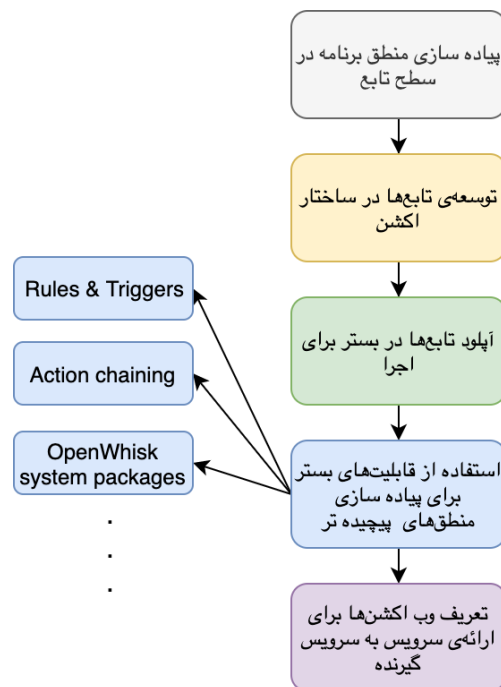
پیاده‌سازی بستر تا اینجا به اتمام می‌رسد اما توسعه‌ی برنامه‌های کاربردی مختلف و دلخواه بر روی این بستر توسط کاربران امکان‌پذیر است. توسعه‌ی این برنامه‌ها در سطح تابع است و مراحل آن عبارت است از تعریف کردن تابع ها به عنوان action های OpenWhisk، استفاده از قابلیت زنجیر کردن action ها، استفاده از سیستم rule و trigger برای ساخت سناریوها و قابلیت‌های دیگر بستر که قبلاً توضیح داده شد.

فصل چهارم

نحوه‌ی کار و ارزیابی

۴-۱ نحوه‌ی کار با بستر بدون سرور اینترنت اشیاء

برای آشنایی با نحوه‌ی کار کردن با بستر بدون سرور اینترنت اشیاء به پیاده‌سازی یک برنامه‌ی کاربردی تست با استفاده از امکانات موجود این بستر می‌پردازیم. این برنامه‌ی کاربردی در دو قسمت سرویس دهنده و سرویس گیرنده پیاده‌سازی شده است. اصلی‌ترین بخش یک برنامه‌ی کاربردی، منطق آن برنامه است که سمت سرور پیاده‌سازی شده و اجرا می‌شود. در این بستر بدون سرور منطق برنامه باید با استفاده از تابع‌ها به عنوان action های بستر OpenWhisk پیاده‌سازی شود. بنابراین منطق کلی برنامه ابتدا باید تا سطح تابع‌های مجزا بخش شود و سپس نحوه‌ی اتصال این تابع‌ها برای ایجاد آن بخش از منطق برنامه تعیین گردد. برای پیاده‌سازی بخش‌های پیچیده‌ی منطق برنامه، می‌توان از امکانات بستر OpenWhisk مانند اتصال تابع‌ها به یکدیگر یا ایجاد rule کمک گرفت. در انتها برای ارائه‌ی سرویس به سرویس گیرنده، تابع‌های مورد نظر به عنوان REST API تعریف شده و سرویس گیرنده می‌تواند با ارسال درخواست‌های HTTP، به سرویس مورد نیازش دسترسی داشته باشد. شکل ۴-۱ مراحل کار با این بستر را نشان می‌دهد.



شکل ۴-۱: مراحل کار با بستر بدون سرور اینترنت اشیاء

برنامه‌ی کاربردی تست در نظر گرفته شده، یک برنامه‌ی کاربردی تلفن همراه است که دستگاه‌های متصل به بستر را نمایش می‌دهد و با انتخاب هر دستگاه، داده‌های ارسالی توسط آن دستگاه به صورت بلادرنگ قابل مشاهده خواهد بود.

۴-۱-۱ سرویس دهنده

برای نمونه نحوه‌ی پیاده‌سازی یکی از action های این برنامه به نام `getTelemetryKeys` را بررسی می‌کنیم (شکل ۴-۲). این action با گرفتن `token` دستگاه، با اتصال به پایگاه داده‌ی سری زمانی، کلیدهای داده‌هایی که آن دستگاه ارسال کرده است را پیدا کرده و برمی‌گرداند. `token` هر دستگاه یک `string` است که مخصوص همان دستگاه است و با استفاده از آن می‌توان داده‌های ذخیره شده در پایگاه داده سری زمانی را برای هر دستگاه استخراج کرد. برای استخراج این داده‌ها از پایگاه داده باید درخواست موردنظر با زبان درخواست `InfluxDB` نوشته شود و سپس با یک درخواست `HTTP` به سرویس دهنده‌ی `HTTP` پایگاه داده ارسال شود. آدرس پایگاه داده درون خوشه‌ی کوبرنتیز، برابر نام سرویس `ClusterIP` آن پایگاه داده است و `port` آن نیز باید مشخص شود. بقیه‌ی تنظیمات این درخواست `HTTP` باید مطابق با `API` پایگاه داده‌ی `InfluxDB` انجام شود. پایگاه داده نتیجه‌ی درخواست را در جواب برمی‌گرداند. در انتها باید این نتیجه توسط action تعریف شده برگردانده شود. این امر به دلیل این که درخواست `HTTP` ناهمگام است به سادگی امکان پذیر نیست، زیرا action باید منتظر جواب درخواست `HTTP` بماند و سپس نتیجه را برگرداند. بنابراین از ساختار `promise` زبان `JavaScript` استفاده شده است. ساختار `promise` این امکان را می‌دهد که action خروجی خود را با تأخیر و به صورت ناهمگام برگرداند.

```

< > getTelemetryKeys.js x
1  const fetch = require('node-fetch');
2
3  function main({ token }) {
4      const q = `SHOW MEASUREMENTS WHERE "token" = '${token}'`;
5
6      return new Promise((resolve, reject) => {
7          fetch(`http://ts-influxdb-host:8086/query?db=tsdb&q=${q}`)
8              .then(response => response.json())
9              .then(response => {
10                 resolve({
11                     data: response.results[0].series[0].values.map(item => item[0])
12                 });
13             })
14              .catch(err => reject(err));
15      });
16  }
17
18  module.exports.main = main;

```

شکل ۴-۲: تابع نوشته شده برای `getTelemetryKeys` action

حال پس از پیاده‌سازی این action، نوبت به اجرا می‌رسد. برای اجرا ابتدا باید کتابخانه‌های مورد نیاز این action نصب شوند. سپس برای ارزیابی، این تابع با استفاده از توابع دیگر که برای تست نوشته شده‌اند به صورت محلی اجرا می‌شود. در صورت موفقیت آمیز بودن ارزیابی، فایل تابع به همراه کتابخانه‌ها، zip شده و با استفاده از اجرای یک دستور در محیط CLI مربوط به OpenWhisk به نام wsk، در بستر آپلود می‌شود. این دستور مورد استفاده در شکل ۳-۴ نشان داده شده است.

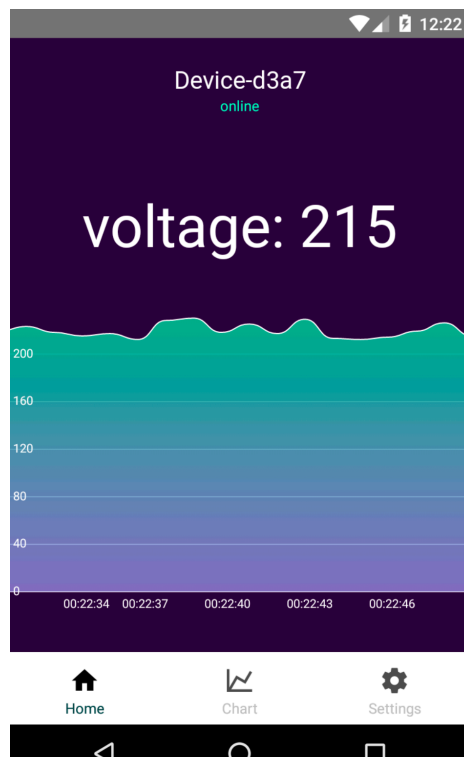
```
(base) Hassans-MacBook-Pro:tok hassanmojab$ wsk -i action create getTelemetryKeys actions/api/getTelemetryKeys.zip
--kind nodejs:default --web true
ok: created action getTelemetryKeys
(base) Hassans-MacBook-Pro:tok hassanmojab$ wsk -i action invoke getTelemetryKeys -p token dummy -br
{
  "data": [
    "current",
    "time",
    "voltage"
  ]
}
(base) Hassans-MacBook-Pro:tok hassanmojab$ wsk -i action delete getTelemetryKeys
ok: deleted action getTelemetryKeys
```

شکل ۳-۴: آپلود، اجرا و حذف action با استفاده از دستور wsk

۲-۱-۴ سرویس گیرنده

سرویس گیرنده‌ی طراحی شده برای این برنامه‌ی تست، یک برنامه‌ی کاربردی تلفن همراه است که با استفاده از فریم وورک React Native پیاده‌سازی شده است. این برنامه‌ی کاربردی از REST API های تعریف شده توسط سرویس دهنده استفاده می‌کند و دستگاه‌ها و داده‌های مربوط به آن‌ها را نمایش می‌دهد. این برنامه یک واسط کاربری گرافیکی^۱ تلقی شده و با مصورسازی داده‌ها برای سهولت درک کارکرد این بستر طراحی شده است. در ادامه تصویری از این برنامه نمایش داده شده است (شکل ۴-۴).

^۱GUI



شکل ۴-۴: برنامه کاربردی تلفن همراه طراحی شده

۲-۴ ارزیابی بستر بدون سرور اینترنت اشیاء

برای اطمینان از صحت کارکرد بستر بدون سرور اینترنت اشیاء، این بستر در بخش‌های مختلف ارزیابی شده و نتایج آن مورد بررسی قرار گرفت. برای ارزیابی بخش سمت دستگاه‌های اینترنت اشیاء باید شبکه‌ای شبیه سازی شده از دستگاه‌های اینترنت اشیاء به بستر متصل شود. برای شبیه سازی دستگاه اینترنت اشیاء، یک اسکریپت نوشته شده است که با اتصال به MQTT message broker از طریق شبکه‌ی اینترنت، هر ثانیه یک پیام MQTT، شامل آن token و داده‌های تولید شده به صورت تصادفی را به بستر ارسال کند. با اجرای چندین نمونه از این اسکریپت با token های مختلف می‌توان شبکه‌ی دستگاه‌های اینترنت اشیاء را شبیه‌سازی کرد. پس از اتصال این شبکه، زمان ارسال پیام توسط دستگاه به عنوان یک داده ارسال شده تا با زمان ذخیره در پایگاه داده مقایسه گردد. اختلاف این دو زمان به طور میانگین برابر با ۳۰۰ میلی ثانیه به دست آمد که با شلوغ شدن شبکه تا ۱۰۰ دستگاه، به ۳ ثانیه افزایش یافت. این زمان برابر است با مجموع تأخیر صف MQTT message broker، زمان پردازش Controller، تأخیر صف Kafka و در نهایت زمان اجرای تابع و ذخیره کردن داده در پایگاه داده.

فصل پنجم

جمع بندی و نتیجه گیری و کارهای آینده

۵-۱ جمع بندی و نتیجه گیری

در این پروژه به طراحی و پیاده سازی یک بستر بدون سرور اینترنت اشیاء پرداخته شد. این کار پس از نیازسنجی در زمینه های اینترنت اشیاء و سرویس های ابری صورت گرفت. همچنین راه حل های مختلف و تکنولوژی های موجود مورد ارزیابی قرار گرفت و راه حل نهایی انتخاب و بررسی شد. در آخر طراحی یک سرویس گیرنده برای آزمایش کردن بستر توضیح داده شد.

همانطور که در فصل های قبل گفته شد، با پیشرفت های اخیر اینترنت اشیاء، استفاده از سرویس های ابری امری ضروری شده است. این سرویس ها می توانند در سطح های مختلفی ارائه شوند و هر کدام برای کاربردهای مختلف مزایا و معایبی دارند. با طراحی و استفاده از بستر بدون سرور اینترنت اشیاء نتیجه گرفته شد که استفاده از این بستر مزایا و برتری های زیر را برای توسعه دهندگان دارد:

۱. سرعت بخشیدن به توسعه نرم افزاری و سخت افزاری و کاهش هزینه ها با حذف نیاز به مدیریت

زیرساخت، مناسب شرکت های نوپا و استارت آپ ها

۲. توانایی ارائه سرویس در سطح نرم افزار برای تیم های سخت افزاری بدون نیاز به داشتن دانش در

حوزه نرم افزار

۳. قابلیت مقیاس پذیری و تعادل بار با توجه به استفاده از تکنولوژی های جدید و معماری میکروسرویس

۴. پیاده سازی منطق برنامه کاربردی در سطح تابع با هر زبان برنامه نویسی

۲-۵ کارهای آینده

کارهای انجام شده در این پروژه مقدمه‌ای بود بر طراحی یک بستر جامع اینترنت اشیاء و بدون سرور. همانطور که در فصل سوم گفته شد، بستر اینترنت اشیاء می‌تواند دارای قابلیت‌های بسیار زیادی باشد که در این پروژه تنها به یک قابلیت آن پرداخته شد. بسیاری از این قابلیت‌ها همچنین ترکیب این قابلیت‌ها با بستر بدون سرور امکان‌های بیشتری به توسعه دهندگان می‌دهد. در ادامه، کارهای آینده برای توسعه بیشتر و بهبود این بستر پیشنهاد می‌شوند:

۱. اضافه نمودن قابلیت کنترل دستگاه‌های اینترنت اشیاء با ارسال دستور به آن‌ها
۲. استفاده از package داخلی بستر OpenWhisk به نام alarms برای ایجاد رویدادها بر اساس زمان
۳. استفاده از package های داخلی دیگر مانند websocket, pushnotifications و ... برای بهبود قابلیت‌های بستر
۴. پیاده‌سازی سیستم‌های logging و monitoring برای بهبود روند توسعه‌ی برنامه‌های کاربردی
۵. فعال‌سازی قابلیت مقیاس پذیری خودکار و پیاده‌سازی در بسترهای ارائه‌دهندگان ابری

منابع و مراجع

- [1] Kubernetes for developers part 2 – replica sets and deployments. <https://nirmata.com/2018/03/03/kubernetes-for-developers-part-2-replica-sets-and-deployments>, March 2018.
- [2] Couchdb. <http://couchdb.apache.org>, 2019.
- [3] Emqx | the leader in open source iot messaging. <https://emqx.io>, 2019.
- [4] Helm. <https://helm.sh>, 2019.
- [5] Influxdb: Purpose-built open source time series database. <https://influxdata.com>, 2019.
- [6] Kubeless. <https://kubeless.io>, 2019.
- [7] Microk8s | zero-ops kubernetes for workstations and edge / iot. <https://microk8s.io>, 2019.
- [8] minikube. <https://github.com/kubernetes/minikube>, 2019.
- [9] Open lambda. <https://github.com/open-lambda/open-lambda>, 2019.
- [10] Openfaas | serverless functions, made simple. <https://openfaas.com>, 2019.
- [11] Baldini, Ioana, Castro, Paul, Chang, Kerry, Cheng, Perry, Fink, Stephen, Ishakian, Vatche, Mitchell, Nick, Muthusamy, Vinod, Rabbah, Rodric, Slominski, Aleksander, and et al. Serverless computing: Current trends and open problems. *Research Advances in Cloud Computing*, page 1–20, 2017.
- [12] Kumar, Manoj. Serverless computing for the internet of things. Master's thesis, Aalto University, Espoo, Finland, 2018.
- [13] Malik, Farhad. What is microservices architecture? <https://medium.com/fintechexplained/what-is-microservices-architecture-1da41a94a29b>, Nov 2018.

- [14] Michele, Sciabarrà. *Learning Apache OpenWhisk: developing open source serverless solutions*. O'Reilly Media, Inc., 2019.
- [15] Pinto, Duarte Pedro, Dias, Joao Sereno, and Ferreira, Hugo undefined. Dynamic allocation of serverless functions in iot environments. *2018 IEEE 16th International Conference on Embedded and Ubiquitous Computing (EUC)*, Jul 2018.
- [16] Roberts, Mike and Chapin, John. *What is Serverless? Understanding the Latest Advances in Cloud and Service-Based Architecture*. O'Reilly Media, Inc., 2017.
- [17] Sanche, Daniel. Kubernetes101: Pods, nodes, containers and clusters. <https://medium.com/google-cloud/kubernetes-101-pods-nodes-containers-and-clusters-c1509e409e16>, May 2018.
- [18] Vaughan-Nichols, Steven J. What is docker and why is it so darn popular? <https://zdnet.com/article/what-is-docker-and-why-is-it-so-darn-popular/>, Mar 2018.

Abstract

With the spread of IoT in research and application areas, the need for IoT cloud services has increased. Most companies and startups that are active in this field also operate only at the hardware and software development level and use cloud providers for data storage and processing. As with Internet of Things, serverless architecture is also evolving in software engineering where the cost and demand of servers is managed by the service provider and only the execution time of the functions is calculated. Also, developers do not need to consider infrastructure and servers conditions. The goal of the project is to set up a serverless platform for executing the functions and run processes of IoT networks, including receiving and analyzing sensor data and executing specific scenarios. This platform can be used by startups, universities, and IoT research centers.

Key Words:

Internet of Things, Serverless, Function as a Service, OpenWhisk, Kubernetes



Amirkabir University of Technology
(Tehran Polytechnic)

Department of electrical engineering

B.Sc. Thesis

Designing and implementing a serverless IoT platform

By

Mohammad Hassan Mojab

Supervisor

Dr. Taheri

September 2019