فصل یک: مقدمه

در محیط محاسبات ابری مقیاس بزرگ، مراکز داده ابر و کاربران نهایی از نظر جغرافیایی در سراسر جهان توزیع شده اند. بزرگترین چالش برای مراکز داده ابر این است که چگونه میلیون ها درخواست را که بطور مداوم از کاربران نهایی می رسند، بطور صحیح و موثر رسیدگی و سرویس داده شوند. برای این منظور پلتفرم هایی در سطح paas که سرویس ابری به شکل پلتفرم است؛ توسعه پیدا کرده اند که می‌توانند این تعداد بار زیاد را مدیریت کنند. یکی از این پلتفرم ها کوبرنتیز(Kubernetes) نام دارد که با مدیریت سرویس ها میتواند بار زیادی را تحمل کند و سرویس های ما را به نحو مطلوبی مدیریت و مقیاس پذیر کند. این پلتفرم به تعادل بار برای رسیدن به بیشترین سطح رضایت کاربر و افزایش نرخ استفاده از منابع محاسباتی و حافظه ای به صورت بهینه کمک می کند.

همچنین این پلتفرم به ما کمک میکند که بتوانیم سرویس های خود را به صورت ساختار مایکرو سرویس در بیاوریم و هر کدام را جداگانه در این پلتفرم بارگذاری کنیم. این روش به ما کمک میکند تا پیچیدگی ها در برنامه نویسی کمتر شود و همچنین بتوانیم سرویس های خود را راحت تر و کاربردی تر مانیتور کنیم و در مواقعی که بار زیاد شد سرویس مورد نظر را مقیاس پذیر کنیم و نه همه ی سرویس های موجود را . این قابلیت همچنین باعث می‌شود در مواقعی که سیستم ما دچار اختلال و مشکل میشود راحت تر ایراد و عیب را پیدا کنند چون عملکرد هر سرویس مشخص است و مشکل راحت تر پیدا میشود.

امروزه شرکت های بزرگ برای مدیریت سرویس های خود از این پلتفرم استفاده میکنند تا خدمات خود را در زمان های پیک باری به نحوی که از دسترس خارج نشوند و زمان تاخیر پاسخشان زیاد نشود؛ ارایه کنند. همچنین این پلتفرم برای سازمان ها و شرکت های بزرگ به این دلیل مهم است که کار خودکارسازی را در ابعاد مختلف انجام میدهد و لازم نیست افراد سازمان ها یا شرکت تا کل روز سرویس ها را مانیتور کنند. به طور مثال هنگامی که بار بر روی یک سرویس زیاد شد ، این پلتفرم سرویس های ما را مقیاس پذیر می‌کند و یا سرویسی از دسترس خارج شد یکی دیگر از آن سرویس اجرا میکند و یا سلامت سرویس ها را مستمر چک میکند و در صورت خطا به ما اطلاع داده میشود. پس با این پلتفرم هزینه مدیریت و نگه داری سرویس ها کاهش پیدا می‌کند و همچنین دسترس پذیری سرویس های ما بیشتر می‌شود که این پلتفرم بسیار از لحاظ اقتصادی و مالی برای شرکت ها به صرفه تر و بهینه تر است‌‌.

هدف از انجام این پروژه مقیاس پذیر کردن سرویس ها در زمان های پیک باری با استفاده از پلتفرم کوبرنتیز در مقیاس کوچک تر است. در ابتدا ما سرویس های خود را به صورت ساختار مایکرو سرویس در می‌آوریم و با استفاده سیستم داکر که در فصل دوم توضیح مختصری از این سیستم ارایه میدهیم؛ سرویس های خود را ایمیج میکنیم که بتوانیم در پلتفرم کوبرنتیز بارگذاری؛ اجرا و مدیریت کنیم. در مرحله بعدی ما معیار های که برای مقیاس پذیری مناسب است را بررسی میکنیم و تعدادی از آنها را معرفی میکنیم. برای آنکه این معیار ها را استخراج کنیم باید سرویس های متفاوتی را در کنار پلتفرم کوبرنتیز خود نصب کنیم. بعد از انجام نصب؛ تست های خود را بر اساس معیار ها انجام میدهیم. دو معیاری که برای ما در این پروژه اهمیت دارد معیار های زمان پاسخ سرویس ها و تعداد درخواستی که در ثانیه می‌توانند جواب دهند. این دو معیار رابطه مستقیم با هم دارند چون اگر زمان پاسخ هر سرویس کمتر شود به تبع آن درخواست بیشتری را می‌توانند جواب دهند و بالعکس. ما با استفاده از این دو معیار کار مقیاس پذیری را انجام میدهیم و سپس علمکرد سرویس های خود را بعد از مقیاس پذیری مورد تحلیل و بررسی قرار می‌دهیم و نتایج را به صورت جدول و گراف به نمایش میگذاریم.

**فصل دوم: مفاهیم ماشین مجازی؛ داکر و کوبرنتیز**

در این فصل به توضیح مفاهیم پایه سیستم داکر و پلتفرم کوبرنتیز می پردازیم. به دلیل آنکه ما سرویس های خود را در این دو محیط پیاده و میکنیم؛ لازم است که با این مفاهیم آشنا باشیم.

# 1 ماشین‌های مجازی و کانتینرها

قبل از ورود به مباحث اصلی داکر، آشنایی با مشخصات ماشین‌های مجازی و مشکلات آنها می‌تواند دلایلی که باعث بوجود آمدن کانتینرها شدند را بیشتر مشخص کند.

## 1-1 ماشین‌های مجازی

قبل از بوجود آمدن کانتینرها، روش اصلی برای ایجاد محیط ایزوله برای مدیریت نرم‌افزارها استفاده از ماشین‌های مجازی بود. در این روش هر نرم‌افزار و سیستم‌هایی که برای اجرا به آنها نیاز داشت در یک ماشین مجازی مستقل با سیستم عامل مجزا نصب می‌شود. چند ماشین مجازی متفاوت می‌توانند بر روی یک سیستم سخت‌افزاری نصب شده و مشخصات این سیستم (مانند فضای هارد یا رم) را به صورت مجزا استفاده کنند. در این حالت در هنگام تعریف هر ماشین مجازی مقدار منابع سخت‌افزاری که توسط ماشین مجازی قابل استفاده است مشخص شده و در صورت نیاز در هر زمان امکان تغییر این موارد وجود دارد.

## 1-1-1 مشکلات ماشین‌های مجازی

الف) ماشین‌های مجازی با توجه به نیاز به نصب کامل سیستم عامل و امکانات مورد نظر هر نرم‌افزار به صورت مستقل، معمولا حجم بالایی از فضای هارد سرور اصلی را اشغال می‌کنند.

ب) راه‌اندازی و اجرای چند ماشین مجازی همزمان نیاز به استفاده از امکانات سخت‌افزاری بالایی را بر روی سرور اصلی ایجاد می‌کند و در غیر این صورت باعث کندی یا ناپایداری سرور خواهد شد.

ج) راه‌اندازی (boot) هر ماشین مجازی به دلیل آنکه باید سیستم عامل مستقل آن ماشین به صورت کامل راه‌اندازی شود مدت زمان زیادی طول می‌کشد.

د) ماشین‌های مجازی به صورت کلی کمک چندانی به انتقال سیستم‌های نرم‌افزاری از یک سرور به سرور دیگر نمی‌کنند و همچنین در صورتی که سیستم عامل‌ها احتیاج به به روز رسانی داشته باشند این کار برای سیستم عامل هر ماشین مجازی باید به صورت مستقل انجام شود.

## 1-2 کانتینرها

کانتینرها بر خلاف ماشین‌های مجازی که ایزوله سازی محیط اجرای نرم‌افزار را در سطح سخت افزار سرور انجام می‌دهند، ایجاد این محیط را به سطح سیستم عامل نصب شده بر روی سرور منتقل می‌کنند. به همین دلیل کانتینرها از نظر استفاده از منابع سرور بسیار کارآمدتر عمل می‌کنند چرا که در این حالت سیستم عامل مجزایی بر روی سرور اصلی نصب نمی‌شود و همینطور منابع نرم‌افزاری مورد نیاز در صورتی که به صورت مشترک در چند کانتینر استفاده می‌شوند می‌توانند بر روی سرور اصلی نصب شده باشند.

### 1-2-1 فواید استفاده از کانتینر

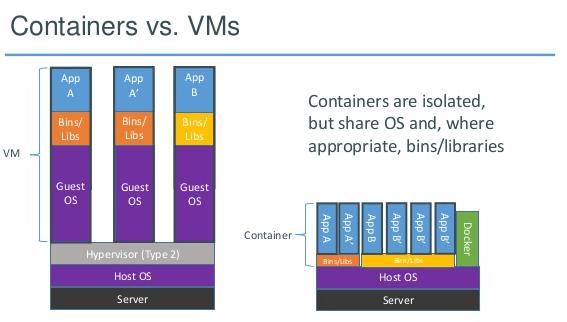
### الف) کانتینرها از سیستم عامل مستقل برای اجرای نرم‌افزارها استفاده نمی‌کنند و می‌توانند منابع نرم‌افزاری مشترک را با هم به اشتراک بگذارند به همین دلیل حجم فضایی که توسط کانتینرها اشغال می‌شود بسیار کمتر از ماشین‌های مجازی است.

### ب) با توجه به اشتراک منابع نرم‌افزار و سخت‌افزاری سرور توسط ماشین‌های مجازی و همچنین عدم نیاز به تخصیص مقدار مشخصی از هر منبع به هر کانتینر، نصب چند کانتینر بر روی یک سرور به امکانات سخت‌افزاری کمتری به نسبت ماشین‌های مجازی احتیاج دارد.

### ج) با توجه به آنکه راه‌اندازی (boot) هر کانتینر فقط نیازمند راه‌اندازی نرم افزارهای مورد نیاز آن کانتینر می‌باشد، سرعت راه ‌اندازی کانتینر بسیار بالاتر از ماشین‌های مجازی است.

### د) انتقال سیستم‌های نرم‌افزار از یک سرور به سرور دیگر به راحتی انجام می‌شود و این سیستم‌ها فقط احتیاج به به‌روز‌رسانی نیازهای نرم‌افزاری مربوط به خود را دارند و به همین دلیل به‌روز‌رسانی کانتینرها به راحتی و تقریبا بدون ایجاد اشکال در کارکرد سیستم امکان‌پذیر است.

در شکل زیر این تفاوت را به شکل واضح تر میتوان مشاهده کرد.



شکل۱ - تفاوت کانتینر و ماشین مجازی

# ۲ داکر(Docker)

داکر ([Docker](https://www.docker.com/)) ابزاری است که برای توسعه، راه‌انداری و اجرای راحت‌تر نرم‌افزارها بوسیله کانتینر (Container) طراحی شده است. کانتینرها به توسعه‌دهنده‌ها اجازه می‌دهند که نرم‌افزارهای خود را به همراه تمام مواردی که برای اجرای آنها احتیاج دارند (کتابخانه‌های نرم‌افزاری و غیره) به صورت یک پکیج آماده کرده و به سرور منتقل کنند. با این روش توسعه‌دهنده می‌تواند مطمئن باشد که نرم‌افزار آماده شده در هر سیستم عاملی که بر روی سرور نصب شده باشد و با هر تنظیماتی که در سیستم عامل ایجاد شده باشد، به درستی کار خواهد کرد و تغییر سیستم عامل یا تنظیمات آن اشکالی در اجرای نرم‌افزار ایجاد نخواهد کرد.

در نگاه اول داکر از نظر کارکرد مشابه ماشین مجازی (virtual machine) به نظر می‌رسد اما بر خلاف ماشین مجازی، بجای راه اندازی یک سیستم عامل کاملا مجزا بر روی سرور، داکر به نرم‌افزارها اجازه می‌دهد که از هسته سیستم عامل اصلی که بر روی سرور نصب شده است استفاده کنند و تنها مواردی که مستقل از سیستم عامل سرور عمل می‌کنند نیازهای اختصاصی نرم‌افزار می‌باشند که بر روی سرور نصب نشده و در کانتینر داکر نصب شده‌اند. این امر افزایش قابل توجهی در عملکرد سیستم ایجاد کرده و حجم کانتینرها را به نسبت ماشین مجازی به مقدار زیادی کاهش می‌دهد.

## 2-1 اجزا اصلی داکر

سه مفهموم اصلی که در هنگام استفاده از داکر باید با آنها کاملا آشنا باشید عبارتند از داکر فایل (Dockerfile)، ایمیج (Image) و کانتینر (Container) مجموعه این مفاهیم، فرایند اصلی داکر و روش استفاده از آن برای مدیریت نرم‌افزارها را مشخص می‌کند.

### **2-1-1 داکر فایل** (Dockerfile)

داکر فایل یک فایل متنی حاوی تمام دستوراتی است که با اجرای آنها تمام نیازمندی‌ها و تنظیمات مربوط به نرم‌افزاری که می‌خواهید توسط داکر اجرا شود در یک بسته داکر به نام داکر ایمیج ایجاد می‌شود. این فایل دستورالعمل مربوط به ایجاد این بسته را در اختیار داکر قرار می‌دهد و داکر با استفاده از این دستورالعمل و با استفاده از دستور docker build این بسته را ایجاد می‌کند.

### **2-1-2 داکر ایمیج** (Docker Image)

داکر ایمیج در تعریف ساده بسته‌ای است که با استفاده از آن می‌توان کانتینرهای داکر را ایجاد کرد. به عبارت دیگر داکر ایمیج بسته‌ای است که پس از ایجاد آن توسط دستورات داکر فایل امکان تغییر آن وجود ندارد و با استفاده از آن می‌توان هر تعداد کانتینر مورد نیاز برای اجرای نرم‌افزار مورد نظر را راه‌اندازی کرد.

داکر ایمیج‌ها در رجیستری داکر ذخیره می‌شوند. این رجیستری می‌تواند یک رجیستری خصوصی باشد که فقط شما به آنها دسترسی دارید یا یک رجیستری عمومی مانند داکر هاب (Docker Hub) که به سایر افراد اجازه می‌دهد از ایمیج ایجاد شده توسط شما برای اجرای نرم‌افزارهای موجود در آن استفاده کنند. استفاده از رجیستری‌های عمومی به صورت متداول در هنگام انتشار نرم‌افزارهای رایگان و سورس باز (Open Source) کاربرد دارند و این امکان را در اختیار دیگران قرار می‌دهند که فقط با دانلود ایمیج و ایجاد یک کانتینر از روی آن به راحتی و بدون نگرانی از نیازمندی‌ها و تنظیمات نرم‌افزار، از آن استفاده کنند.

### **2-1-3 داکر کانتینر** (Docker Container)

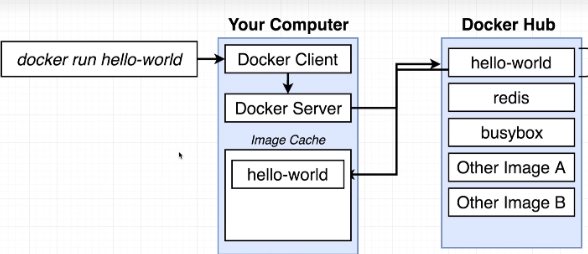
با استفاده از داکر ایمیج و دستور docker run می‌توان یک نمونه‌ی اجرایی از نرم‌افزار مورد نظر را به صورت داکر کانتیتر ایجاد کرد. بنابراین داکر کانتینر به صورت کلی یک نسخه‌ی آماده‌ی اجرا از نرم‌افزار موجود در داکر ایمیج هستند که هدف نهای استفاده از داکر و اجرای نرم‌افزار توسط آنها را محقق می‌کنند.



شکل۲ - نحوه کارکرد سیستم داکر

## 2-2 کارکرد کلی پلتفرم داکر

داکر از دو بخش داکر کلاینت و داکر سرور تشکیل شده است که بخش کلاینت دستورات را از ما میگیرد و به داکر سرور برای اجرای آن دستور می فرستد. داکر کلاینت به CLI سرور ما متصل شده است که از آن طریق دستورات را دریافت میکند. با استفاده از داکر سرور ما میتوانیم ایمیج های خود را با استفاده داکر فایل ها درست کنیم و به داکر هاب منتقل کنیم. این کار باعث میشود در هر سرور دیگری بتوانیم این ایمیج ها را به راحتی اجرا کنیم. حال در شکل زیر این ارتباط را مشاهده میکنیم.

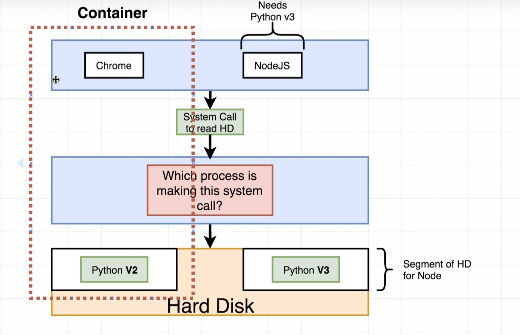


شکل۳ - نحوه اجرای یک ایمیج از داکر هاب

همان طور که در شکل بالا مشاهده میشود با اجرای دستور داکر ما به داکر کلاینت متصل میشویم و سپس اگر ایمیج قبلا از داکر هاب دانلود شده بود؛ دیگر نیازی به متصل شدن به داکر هاب نیست و داکر سرور این ایمیج را اجرا خواهد کرد. در غیر این صورت باید متصل شود و ایمیج را دانلود و سپس اجرا کند.

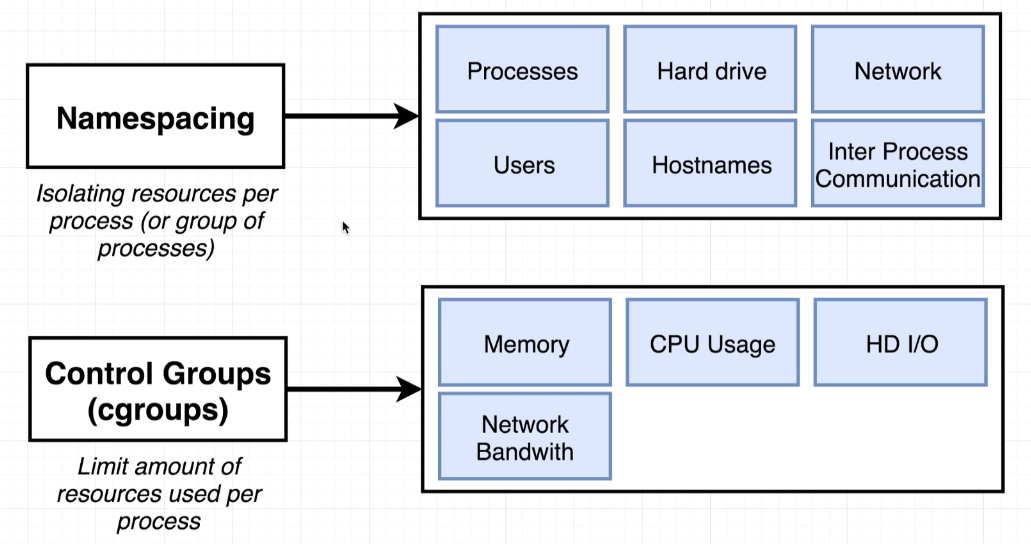
## 2-3 نحوه اجرا شدن کانتینر در سرور

همان طور که گفته شد کانتینر ما یک نرم افزار است که با تمام مواردی لازم دارد اجرا شود ایزوله میشود و توسط container runtime اجرا میشود. حال میخواهیم ببینیم این عملیات چگونه صورت میپذیرد.



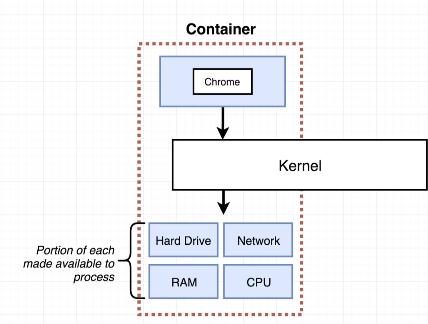
شکل۴ - فضای یک کانتینر در سرور

در شکل زیر مشاهده میکنیم برای اینکه این برنامه اجرا شود به Python V2 نیاز دارد. پس قسمتی از دیسک به این منظور برای این برنامه جدا میشود و در آن این برنامه ذخیره میشود. پس کانتینر به این صورت عمل میکند که یک برنامه ای است که نیازمندی های نرم افزاری و سخت افزاری اش توسط سیستم عامل مدیریت میشود و مفهومی به اسم namespace وجود دارد که برای هر کانتینر یک فضای مختص از لحاظ حافظه؛ شبکه ای و موارد دیگر در نظر گرفته میشود که هر موقع این برنامه درخواستی از سیستم عامل داشت؛ سیستم عامل تشخیص بدهد کدام برنامه است و هنگامی که این برنامه نیاز به این برنامه ها برای اجرا شدن داشت باید به کدام قسمت حافظه باید درخواست بدهد.



شکل۵ - Namespacing و cgroups در داکر

در شکل بالا مشاهده می شود که با استفاده از namespacing داکر سرور ما محیط ایزوله برای منابع کانتینر ما فراهم میکند و دسترسی کانتینر به آن namespace محدود میشود. Cgroup ها هم مشخص میکنند که هر برنامه ما که به صورت کانتینر در آمده است چه مقدار اجازه دارد که از منابع مختلفی که در سرور وجود دارد و در شکل بالا هم برخی از این منابع آمده است استفاده کند. شکل زیر به صورت ملموس تر نشان میدهد که یک کانتینر در سیستم ما چگونه است. با استفاده از درخواست هایی که برنامه ما که همان یک پروسس در سیستم عامل ما هست به کرنل میدهد؛ سیستم عامل با استفاده از منابعی که برای این پروسس در نظر گرفته است جواب میدهد و منابع را در اختیار این پروسس قرار میدهد.



شکل۶ - نحوه ایزوله شدن یک کانتینر در داکر

## 2-4 میکرو سرویس ها

بعد از اینکه سیستم داکر شهرت گرفت؛ این پلتفرم برنامه نویسان را به این سمت سوق داد که برنامه های خود را به صورت مایکروسرویس بنویسند به این معنی که سرویس خود را به بخش های کوچکتر تقسیم میکنیم و هر کدام وظیفه انجام کاری را دارند. این نوع ساختار برنامه نویسی باعث شد تا برنامه هایی که پیچیدگی زیاد دارند را با تقسیم به برنامه های کوچکتر این پیچیدگی کمتر شود و راحت تر و سریع تر بتوانند کار عیب یابی و رفع اشکالات را انجام دهند. همچنین با استفاده از این پلتفرم میتوان از ارکستریتور ها هم استفاده کرد به این منظور که هنگامی که بار بر روی یک کانتینر زیاد شد تعداد بیشتری از این کانتینر ها ساخته شود و سرویس دهی بهبود یابد.

در این پروژه ما از این ساختار استفاده خواهیم کرد و سرویس های خود را به مایکرو سرویس ها در خواهیم آورد تا بتوان بر اساس پلتفرم کوبرنتیز که در بخش بعدی به آن خواهیم پرداخت؛ این مایکرو سرویس ها را مدیریت کنیم.

# 3 پلتفرم کوبرنتیز (Kubernetes)

## 3-1 توضیح پلتفرم کوبرنتیز و مزیت های استفاده از آن

کوبرنتیز ( Kubernetes) (که به شکل k8s نیز ارجاع می‌شود) سامانه‌ای متن‌باز برای خودکارسازی دیپلوی(deployment)؛ مقیاس و مدیریت برنامه‌های کانتینرسازی شده در سراسر زیرساخت است که در ابتدا توسط گوگل توسعه داده شد و سپس در سال ۲۰۱۵ به بنیاد CNCF اهدا شد. کوبرنتیز پیاده سازی جدیدی از بیش از یک دهه تجربه گوگل در اجرای نرم افزارهای سمت سرور در مقیاس بسیار بالاست که به صورت متن باز (open source) در اختیار همه قرار گرفته است.

این پلتفرم وظیفه اجرا و مدیریت کانتینرها را بر روی گروهی از سرورهای موجود در یک یا چند مرکز داده ها (data center) به عهده دارد. کوبرنتیز در واقع نسل سوم از این فناوریست که در شرکت گوگل از ابتدا به زبان گو (Go) پیاده سازی شده است. دو نسل قبلی آن برگ (Borg) نام داشته که پیاده سازی آن به زبان سی پلاس پلاس بوده است و گوگل همچنان از آن در محیط عملیاتی استفاده میکند.

مزیت کلیدی کوبرنتیز در این است که بدون نیاز به یک تیم بزرگ برای راه اندازی و نگهداری، میتوان آن را در مقیاس وسیع برای اجرای تعداد زیادی برنامه کاربردی به کار گرفت. از مزایای دیگر آن قابلیت اجرا بر روی بسترهای متفاوت است؛ از سرورهای یک مرکز داده های خصوصی گرفته تا سرویسهای ابری عمومی، یا حتی ترکیبی از هر دو. به طور کلی هر شرکتی که یک یا چند سرویس نرم افزاری اجرا می کند به طور بالقوه در مرحله اول به کانتینرها و سپس به سیستمی مانند کوبرنتیز نیاز دارد. دلیل اصلی نیاز به کانتینرها امکان جداسازی برنامه ها (isolation) از یکدیگر در بهترین سطح ممکن است تا فرآیند تولید، تست و در نهایت اجرا بر روی یک زیرساخت مشترک تسهیل شود.

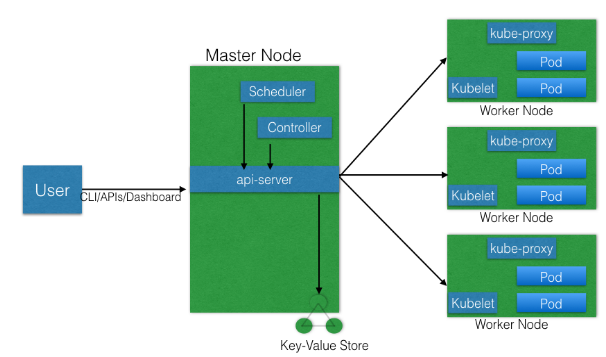
در مرحله بعد نیاز به کوبرنتیز پیدا میشود تا اجرای این کانتینرها بر روی دسته ای (cluster) از ماشینها را تا حد زیادی اتوماتیک کند. در واقع کوبرنتیز مانند سیستم عاملیست که بر روی تمام سرورهای شما به صورت یکپارچه اجرا میشود و به شما این امکان را میدهد که دیگر نگران هیچ ماشینی به طور خاص نباشید. اگر ظرفیت کافی در زیرساخت شما وجود داشته باشد، این سیستم به راحتی میتواند از دست دادن یک یا چند ماشین را برای شما به گونه ای مدیریت کند که کاربران هیچ تغییری در سرویسهای در حال اجرا بر روی این بستر احساس نکنند.

این سیستم امکاناتی مانند بررسی سلامت (health check) و تکثیر (replication) برنامه ها را به راحتی بر روی مجموعه سرورهای شما فراهم میکند. از دیگر قابلیتهای آن نیز ویژگیهای مناسب و سطح بالا، مانند کشف سرویسها (service discovery)، توزیع بار (load balancing) و مدیریت پیکربندی (configuration management) است که برای ساخت سیستمهایی با معماری مایکروسرویسی (micro-service architecture) حیاتیست و برای تیمهای شما امکان تولید، تغییر و مقیاس پذیری (scaling) بخشهای مختلف هر سرویس را بر اساس شرایط مورد نیاز فراهم میکند. همچنین این سیستم به صورت خودکار کانتینر هایی که خراب شده اند را از بین میبرند و دوباره تولید میکنند و اجازه اینکه ترافیک به این کانتینر خراب فرستاده شود؛ جلوگیری میکند. مزیت دیگری که میتوانیم برای این سیستم نام ببریم؛ مدیریت منابع حافظه ای است که میتواند حافظه برای کانتینر ها را از طرق مختلف تامین کند. به طور مثال از طریق حافظه داخلی سرور یا حافظه فراهم کنندگان ابری و یا از طریق Network Storage System فراهم کند. در کنار برخی از مزایایی که برای این پلتفرم گفته شد؛ قابلیت کاربردی و مهمی که این سیستم دارد این است که میتوان این پلتفرم را توسعه داد و بخش هی متفاوتی اضافه کرد. این به این دلیل است که ساختار کوبرنتیز هم ماجولار (Modular) است و توسعه دهندگان می توانند بخش های مختلفی را به این پلتفرم اضافه کنند.

اگر چه بسیاری از نرم افزارها سعی میکنند این قابلیت ها را در سطح برنامه کاربردی پیاده کنند ولی تجربه نشان داده است که این کار با وجود صرف زمان و انرژی زیاد در اکثر موارد منجر به یک راه حل شکننده و غیر قابل نگهداری میشود که برای برنامه های کاربردی بعدی باید از نو تکرار شود. کوبرنتیز با انتقال این دغدغه ها به لایه مناسب و آزاد کردن برنامه کاربردی از قید و بند آنها به شما کمک میکند که وقت و انرژی تیم را در جای مناسب و برای تولید ویژگیهای خاص برنامه کاربردی خودتان صرف کنید.

## 3-2 اجزا و ساختار پلتفرم و گره رهبر در کوبرنتیز

حال در این بخش به ساختار و اجزای کوبرنتیز می پردازیم و بخش های مختلف این پلتفرم را توضیح میدهیم تا دید بهتری نسبت به این پلتفرم پیدا کنیم.

شکل۷ - ساختار کلی پلتفرم کوبرنتیز

همان طور که در شکل بالا مشاهده می کنیم؛ پلتفرم کوبرنتیز شامل چند بخش مهم است که اصلی ترین بخش آن گره رهبر (Master node) است. این بخش مسئول مدیریت و هماهنگی بخش های مختلف کلاستر ما است و به نوعی همه عملیات های کلاستر ما از طریق این بخش انجام میشود. این گره شامل بخش های متفاوتی است که در بخش زیر آن ها را توضیح خواهیم داد. همه کاربران برای آنکه بتوانند به کلاستر دسترسی پیدا کنند و عملیات های مختلف انجام دهند باید از طریق این گره درخواست های خود را بفرستند. این درخواست ها هم از طریق ترمینال یا داشبورد و یا API های مختلف امکان پذیر است.

به دلیل آنکه این گره بسیار نقش اساسی در کلاستر ما دارد؛ باید همیشه در دسترس باشد و اگر دچار خطا شود؛ هزینه زیادی را به شرکت یا سازمان تحمیل خواهد کرد. برای رفع این مشکل؛ معمولا چند گره رهبر به کلاستر اضافه میکنند و به عبارتی کلاستر در حالت دسترس پذیری بالا (HA)قرار میدهند و اگر یکی از این گره ها دچار مشکل شد؛ گره های رهبر بتوانند سرویس دهی را انجام دهند. حال سراغ بخش های مختلف این گره می رویم و توضیح مختصری برای هر کدام ارائه میکنیم.

الف) API server

این نقطه‎ی اصلی مدیریت و ورود به Kubernetes است که به کاربر اجازه می‎دهد Kubernetes را پیکربندی کند. در واقع kube-apiserver پلی بین اجزای مختلف با هدف نگهداری و حفظ سلامت کلاستر و انتشار اطلاعات و اجرای دستور عمل‌ها است. api server یک رابط RESTfull ایجاد می‎کند که به این معنی است که بسیاری از ابزارها و کتابخانه‎ها به راحتی می‎توانند با آن ارتباط برقرار کنند. api server درخواست ها را دریافت میکند؛ سپس آنها را تایید و بررسی میکند. بعد از آنکه حالت فعلی کلاستر را از یک پایگاه داده توزیع شده به اسم etcd که در بخش بعد توضیح خواهیم داد؛ خواند و با نتیجه درخواستی که الان آمده است مقایسه کرد؛ حالت جدید تولید شده از نتیجه درخواست فعلی را در این پایگاه داده ذخیره می کند. api server تنها بخشی است که اجازه دارد به این پایگاه داده دسترسی پیدا کند و اطلاعات کلاستر ما را بخواند یا بنویسد. همچنین در پلتفرم کوبرنتیز این امکان وجود دارد که چند api server وجود داشته باشد و api server اصلی ما درخواست ها را به بقیه api server ها بفرستد و آن ها را مدیریت کند.

ب)Scheduler

این بخش مسئولیت این را دارد که آبجکت ها جدید مانند پاد ها را به گره ها اختصاص بدهد. برای آنکه scheduler ما بتواند تصمیم بگیرد که کدام آبجکت را به کدام گره اختصاص دهد؛ ابتدا باید نیازمندی ها و محدودیت های هر آبجکت را که در کانفیگ فایلش نوشته میشود بررسی کند و همچنین ظرفیت گره های کارگر را بررسی کند. scheduler این اطلاعات را باید از پایگاه داده etcd از طریق api server دریافت کند و سپس پس از بررسی این اطلاعات کار اختصاص دادن این آبجکت ها به گره ها را انجام میدهد و حالت فعلی کلاستر را از طریق api server در etcd ذخیره میکند. این بخش هم همچنین قابل توسعه است و توسعه دهندگان میتوانند scheduler های خود را به کوبرنتیز اضافه کنند. برای آنکه آبجکت ما توسط یک scheduler خاص زمانبندی شود باید در کانفیگ فایل آن آبجکت اسم scheduler را گذاشته باشیم در غیر این صورت توسط scheduler پیش فرض زمانبندی میشود.

ج)Controller managers

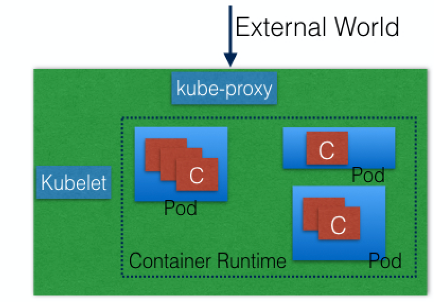
این کنترلر ها مسئولیت این را دارند که وضعیت فعلی کلاستر مطابق وضعیت مطلوب باشد. به طور مثال اگر در کانفیگ فایل نوشته شده است که باید ۳ پاد اجرا شود و در حال حاضر ۱ پاد برای یک آبجکت اجرا میشود؛ باید تعداد پاد ها را زیاد کند تا وضعیت مطلوب حاصل شود. این کنترلر وضعیت فعلی را از پایگاه داده etcd از طریق api server دریافت میکند و به طور مستمر این مقایسه را انجام میدهد که کلاستر ما در وضعیت مطلوب نگه داشته شود.

د)etcd

این بخش یک پایگاه داده توزیع شده است که داده ها را به صورت key-value نگه داری میکند. اطلاعات کانفیگ فایل های آبجکت های مختلف در این پایگاه داده ذخیره میشود که تنها api server میتواند به این پایگاه داده دسترسی پیدا کند و اطلاعات را بخواند و بنویسد. این پایگاه داده هم میتواند در گره رهبر ما قرار بگیرد و یا آنکه جدا از این گره باشد و در گره های دیگری قرار بگیرد به منظور آنکه اگر گره رهبر ما دچار مشکل شد یا مورد حمله قرار گرفت؛ اطلاعات کلاستر ما از بین نرود. همچنین کوبرنتیز قابلیت های backup ؛ snapshot و بازیابی اطلاعات را برای این پایگاه داد قرار داده است تا اطلاعات ما به راحتی از بین نرود.

## 3-3 معماری و اجزا تشکیل دهنده گره کارگر

بعد از اتمام بخش گره رهبر ما سراغ گره کارگر میرویم و بخش های مختلف این گره را بررسی میکنیم.



شکل۸ - شمای کلی گره کارگر در پلتفرم کوبرنتیز

سرویس های ما در این گره های کارگر اجرا میشوند و سرویس های ما در آبجکت پاد که کوچکترین آبجکت پلتفرم کوبرنتیز است؛ قرار دارند که هر پاد میتواند چند کانتینری که عملکردشان بسیار مرتبط با هم است قرار بگیرند که اکثرا در هر پاد یک کانتینر اجرا میشود. کاربران با دسترسی به این گره ها میتوانند به سرویس های ما دسترسی پیدا کنند و عملیات مورد نظر را انجام دهند. در فصل ۳ انواع آبجکت های کوبرنتیز را مورد بررسی قرار میدهیم.

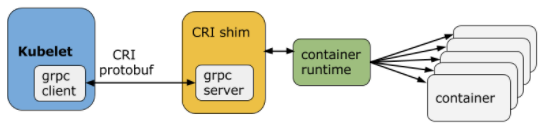
همانطور که در شکل بالا مشاهده می کنیم؛ گره کارگر ما دارای سه بخش Container Runtime ؛ Kube-proxy و Kubelet است. در بخش زیر توضیح مختصری برای هر کدام ارائه میدهیم.

الف) Container Runtime

اگرچه همانطور که گفتیم کوبرنتیز یک پلتفرم برای مدیریت و اجرای کانتینر ها است؛ ولی این پلتفرم خود به طور مستقیم این کانتینر ها را نمیتواند مدیریت و اجرا کند. برای اینکار باید از یک Container Runtime برای هر گره کارگر استفاده کند تا کوبرنتیز بتواند پاد ها را در آن گره اجرا و مدیریت کند. Container Runtime های متفاوتی مانند Docker؛ Containerd و CRI-O وجود دارد که ما در این پروژه از Docker برای اجرای کانتینر ها استفاده میکنیم.

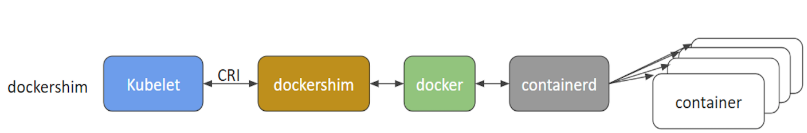
ب)Kubelet

این عامل در گره کارگر مسئولیت ارتباط با بخش های مختلف گره رهبر را دارد و کانفیگ فایل پاد را از طریق api server دریافت میکند سپس به container runtime وصل میشود و کانتینر ها مشخص شده در کانفیگ فایل را از طریق container runtime اجرا میکند. این عامل همچنین سلامت کانتینر هایی که در حال اجرا هستند را مانیتور میکند.



شکل۹ - نحوه ارتباط kubelet با container runtime

همانطور که در شکل بالا مشاهده میکنیم؛ kubelet از طریق Container Runtime Interface توانسته است به Container Runtime متصل شود. این CRI شامل پروتکل هایی است که Kubelet از طریق آن ها میتواند به Container Runtime متصل شود. در اینجا Kubelet به عنوان grpc client عمل میکند و به CRI shim که به عنوان grpc server عمل میکند؛ متصل میشود و عملیات های مربوط به کانتینر ها را انجام میدهد. CRI دو سرویس اصلی دارد. اولی به عملیات های مربوط به ایمیج ها میشود که ImageService نام دارد و دومی عملیات های مربوط به کانتینر ها و پاد ها میشود که RuntimeSerivce نام دارد. همانطور که قبلا هم اشاره کردیم؛ کوبرنتیز با container runtime های مختلفی میتواند کار کند به شرط آنکه CRI را داشته باشد.



شکل۱۰ - نحوه ارتباط kubelet با محیط اجرای کانتینر برای docker

به دلیل آنکه ما از داکر برای اجرای کانتینر های خود استفاده میکنیم؛ ارتباط kubelet با داکر به صورت شکل بالا است و از طریق CRI مخصوص داکر به داکر متصل میشود و داکر هم از طریق containerd کانتینر ها را اجرا میکند.

ج) kube-proxy

این عامل ارتباطاتی در گره کارگر است که IP های پاد ها در IPtables نگه داری میکند و هنگامی که این IP ها تغییر کرد باید جدول را بروزرسانی کند. همچنین درخواست هایی که از طرف کاربران به این گره وارد میشود را به پاد ها می فرستد. در فصل ۳ بیشتر به نحوه ارتباطات پاد ها با یکدیگر و محیط بیرون میپردازیم.

# 4 جمع بندی

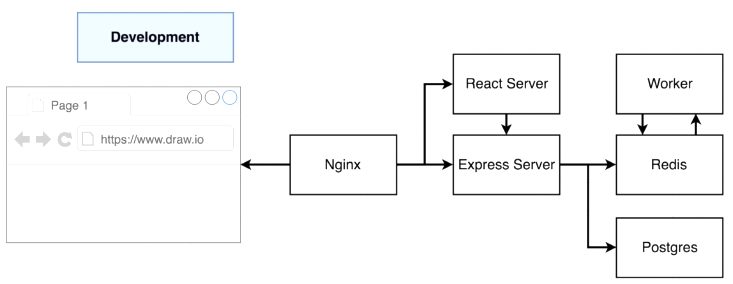
در این فصل؛ مزیت های کانتینر ها نسبت به ماشین های مجازی گفته شد و همچنین محیط داکر به منظور اجرا شدن کانتینر ها معرفی و توضیح داده شد. در بخش آخر ما پلتفرم کوبرنتیز و اجزای مختلف آن را توضیح دادیم و نحوه بارگذاری و اجرای داکر بر روی این پلتفرم بیان گردید. دلیل توجه ما به این موضوع این است که ما قصد داریم سرویس های خود را به صورت ایمیج درآوریم و از طریق داکر آن ها را اجرا کنیم و با استفاده از پلتفرم کوبرنتیز مدیریت خودکار بر روی این کانتینر ها داشته باشیم. عملکرد گره های رهبر و کارگر نیز در محیط کوبرنتیز در این رابطه تشریح گردید.

فصل سوم : توضیح سرویس ها و ایمیج کردن مایکرو سرویس ها با استفاده از سیستم داکر

1 توضیح مایکرو سرویس ها و ساختارشان:

در این فصل ما قصد داریم تا سرویس های خود را به صورت مایکرو سرویس ها دربیاوریم تا بتوانیم از مزایای این ساختار بهره ببریم و پیچیدگی سیستم خود را کمتر کنیم و همچنین بتوانیم این سرویس ها را در پلتفرم کوبرنتیز بارگذاری کنیم. بعد از اتمام پیاده سازی سرویس ها، باید با استفاده از سیستم داکر ایمیج های این سرویس ها را بسازیم و در داکر هاب بارگذاری کنیم به منظور اینکه بتوانیم در هر نود های مختلف از این سرویس‌ها استفاده کنیم.

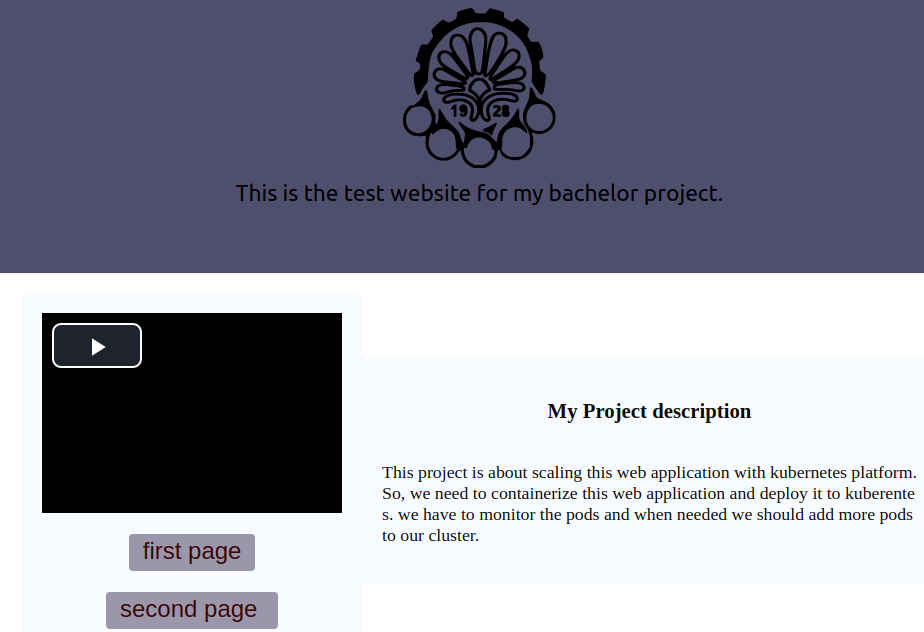
برای این پروژه ما یک سرویس کوچک و ساده را پیاده سازی میکنیم که پیچیدگی سیستم زیاد نشود و بیشتر بتوانیم در ابعاد مدیریت و مقیاس پذیری این سرویس ها کار کنیم. برای این منظور سرویس محاسبه فیبوناچی را مد نظر گرفته ایم که این سرویس را به چند مایکرو سرویس تقسیم بندی میکنیم. سیستم خود را به شیش مایکرو سرویس تقسیم بندی کرده ایم. یک فرانت اند و بک اند وجود دارد که فرانت اند (client) با زبان react است و بک اند(server) آن با استفاده از فریم ورکی از node js به نام express نوشته شده است. به علاوه بر این دو یک مایکرو سرویس دیگر هم به اسم worker وجود دارد که کار محاسبه کردن فیبوناچی را انجام میدهد. و دو پایگاه داده مجزا هم داریم که یکی برای ذخیره اعداد وارد شده از طرف کاربر استفاده میشود و دیگری برای ذخیره اعداد محاسبه شده از طرف سرویس worker . هدف از انتخاب دو پایگاه داده مجزا به این منظور بود که سرویس های بیشتری داشته باشیم و هدف خیلی خاصی برای این انتخاب وجود نداشت. و سرویس آخر هم وب سرور ما است که درخواست های وارد شده از طرف کاربران را به سرویس های مختلف می فرستد و سرویس های ما فقط از این طریق به دنیای بیرون می‌توانند اتصال یابند. البته برای آنکه ما سرویس های خود را از طریق اینترنت در دسترس قرار دهیم باید یک لود بالانسر (Load balancer) از فراهم کننده های ابر خریداری کنیم و وب سرور ما به آن لود بالانسر متصل شود که بتوانیم یک IP address خارجی داشته باشیم و بقیه کاربران از این طریق به سرویس های ما متصل شوند. ولی در این پروژه ما فقط از وب سرور استفاده میکنیم و به صورت محلی بار تولید خواهیم کرد. در شکل زیر هم ارتباطات مایکرو‌ سرویس های مختلف باهم را مشاهده میکنیم.



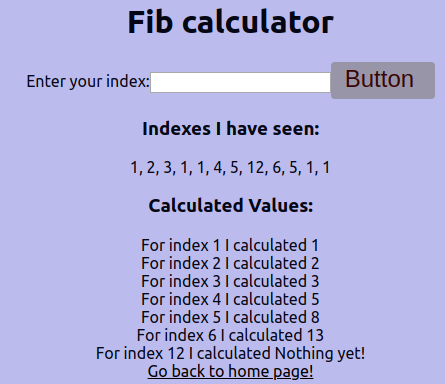
شکل ۱ - نحوه ارتباطات مایکرو سرویس ها با یکدیگر

عملکرد کلی سیستم ما به این صورت است که کاربر داده را در فرانت اند وارد میکند و مایکرو سرویس server که مسئول بخش بک اند است، داده را هم در پایگاه داده postgres و هم در پایگاه داده redis میگذارد. که پایگاه داده اولی فقط به این منظور است که به کاربر نشان دهیم تا به حال چه اعدادی را وارد کرده است و دومی برای محاسبه کردن این عدد توسط مایکرو سرویس worker است. مایکرو سرویس worker، داده را از پایگاه داده redis بر میدارد، میخواند و محاسبه میکند و در آخر نتیجه توسط سرویس server از پایگاه داده خوانده میشود و از طریق سرویس client که مسئول پخش فرانت اند است، به کاربر نمایش داده میشود. و در اخر با استفاده از مایکرو سرویس nginx روتینگ را انجام میدهیم و هر در خواست را مطابق با سرویسی که درخواست شده است به مایکرو سرویس مد نظر منتقل میکند.

شمای فرانت اند که سرویس client ما است به این شکل زیر است که دارای دو صفحه است. در صفحه دوم سرویس فیبوناچی ما مشخص است که شمای این سرویس هم بعد از شکل فرانت اند قرار گرفته است. همان طور که در شکل دوم صفحه بعدی معلوم است؛ نتایج سرویس های مختلف خود را مشاهده میکنیم. در قسمت اول نتایج سرویس پایگاه داده postgres ما است که اعدادی که تا به حال کاربر وارد کرده است نشان داده میشود که در این بخش سرویس های client؛ server و postgres دخیل بوده اند. در قسمت دوم نتایج اعداد وارد شده توسط سرویس worker محاسبه میشود و نمایش داده میشود.



شکل ۱ - شمای صفحه اول سرویس client

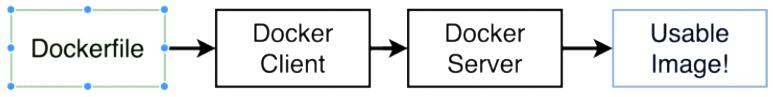


شکل ۲ - نتایج مایکرو سرویس های server؛ client؛ worker؛redis و postgres

۲ ایمیج کردن مایکرو سرویس ها:

قبل از آنکه ما مایکرو سرویس های خود را ایمیج کنیم؛ ابتدا توضیحاتی در مورد مراحل ایمیج درست کردن را ارائه می دهیم.

برای آنکه بتوانیم ما ایمیج خود را بسازیم باید یک کانفیگ فایل به اسم Dockerfile را درست کنیم که مشخص میکند کانتینر (container) ما چه برنامه هایی را شامل میشود و چه دستوری را موقع اجرا شدن باید استفاده کند. سپس این کانفیگ فایل را از طریق ترمینال به داکر کلاینت (docker client) میدهیم و داکر کلاینت هم این فایل را به داکر سرور(docker server) میدهد. داکر سرور همه خط های این فایل را میخواند و ایمیج مد نظر ما را میسازد. در شکل زیر مراحل ساخت ایمیج را مشاهده میکنیم.



شکل ۳ - مراحل ساخت ایمیج (Image)

2-1 Dockerfile

در این بخش سراغ دستوراتی که باید در این کانفیگ فایل بنویسیم میرویم و توضیحاتی درباره هر کدام میدهیم. این کانفیگ فایل دارای سه بخش است:

الف) مشخص کردن base image

ب) اجرا کردن دستوراتی برای نصب برنامه هایی که در کانتینر میخواهیم استفاده کنیم

ج) مشخص کردن دستوری که هنگامی که کانتینر اجرا میشود باید آن دستور را اجرا کند

به منظور آنکه این کانفیگ فایل را بهتر توضیح بدهیم؛ مراحل را از روی ایمیج کردن یکی از سرویس هایی که در این پروژه استفاده کرده ایم به نام پایگاه داده redis میرویم. ایمیج پایگاه داده redis در داکر هاب (Docker Hub)موجود است و ما در پروژه از این ایمیج استفاده میکنیم ولی برای درک بهتر این پروسه؛ dockerfile این سرویس را توضیح میدهیم.



شکل ۴ - Dockerfile سرویس redis

داکر سرور با خواندن این کانفیگ فایل دستوراتی که برای درست کردن یک ایمیج لازم است را میگیرد و ایمیج مد نظر ما را میسازد. دستورات دیگری هم برای dockerfile وجود دارد ولی این دستورات؛ دستورات کلی و مهم برای ساختن یک ایمیج هستند.

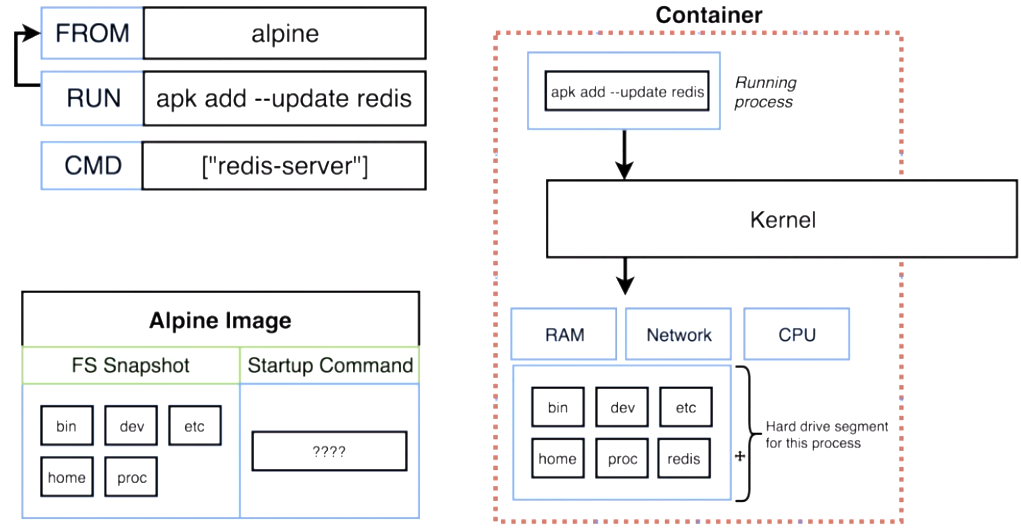
دستور اول به این منظور است که ما برای اینکه ایمیج خود را درست کنیم ابتدا به یک ایمیج پایه نیاز داریم که بتوانیم برنامه ها و وابستگی های مختلف را دانلود و نصب کنیم. این ایمیج پایه شامل برنامه های مختلفی است که ما با استفاده از آن میتوانیم ایمیج خود را بسازیم و بدون این ایمیج پایه امکان درست کردن ایمیج مد نظر خود را نداریم. این دستور مانند این است که ما میخواهیم نرم افزاری را در کامپیوتر خود نصب کنیم. برای این کار ابتدا یک سیستم عامل نیاز داریم تا بتوانیم نرم افزار خود را دانلود و نصب کنیم و سپس اجرا کنیم. پس این ایمیج پایه مشابه سیستم عامل در کامپیوتر عمل میکند.

با استفاده از دستور دوم ما برنامه ای را که میخواهیم دانلود و نصب میکنیم. در اینجا ما سرویس redis را میخواهیم دانلود و نصب کنیم که به استفاده از پکیج منیجر apk که در ایمیج پایه alphine نصب شده است این کار امکان پذیر است. در آخر هم با استفاده از دستور CMD مشخص میکنیم هنگامی که کانتینر ما اجرا شد برنامه redis را درون این کانتینر اجرا کند.

بعد از این توضیحات سراغ ساختن ایمیج مد نظر میرویم که میتوانیم با استفاده از دستور زیر این کار را انجام دهیم .

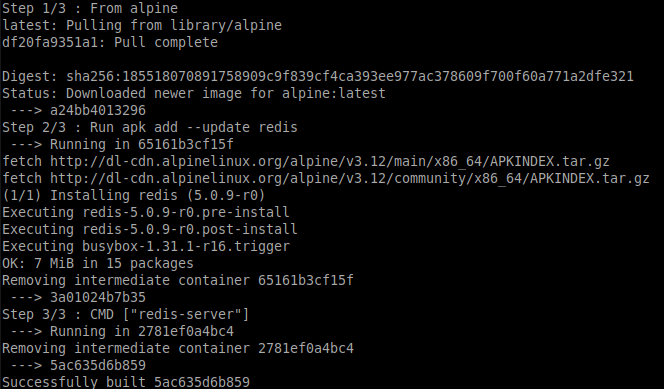
"docker build . "

توجه داشته باشیم که نقطه بعد کلمه build حتما آورده شود تا مراحل ساختن ایمیج کردن شروع شود. و این نقطه به این معنی است که در همین مسیری که الان در آن هستیم و dockerfile ما وجود دارد اشاره میکند. حال توضیحی در مورد مراحل ساخته شدن ایمیج توسط داکر سرور میدهیم.



شکل ۴ - کانتینر های میانی برای ساخته شدن نهایی ایمیج مد نظر

هر ایمیجی یک فایل سیستم و یک دستور هنگامی که کانتینر شروع به اجرا شد؛ دارد که در شکل بالا این ساختار را در شکل پایین سمت چپ مشاهده می کنیم. در مرحله اول داکر سرور ایمیج پایه ما را دانلود میکند و دستور" apk add --update redis" را در کانتینر جدید ساخته شده اجرا میکند که این کانتینر میانی اول برنامه ها را دانلود و نصب میکند. بعد از این عملیات ایمیج جدید تولید شده و این کانتینر میانی اول حذف می شود. ایمیج جدید شامل برنامه های ایمیج پایه و برنامه redis است. در مرحله اخر این کانتینر میانی دوم با دستور "redis-server" اجرا میشود و بعد از ذخیره کردن دستور جدید و برنامه های جدید؛ ایمیج مرحله پایانی ساخته میشود و کانتینر میانی دوم هم حذف میشود. در شکل زیر این مراحل را با جزئیات بیشتری می توانیم مشاهده کنیم که Id کانتینر های میانی و دستورات آمده است.



شکل ۵ - مراحل ساخته شدن ایمیج نهایی

همانطور که در شکل بالا مشاهده می کنیم؛ اسم ایمیج ما یک عدد است که در اینجا 5ac635d6b859 است که به یاد داشتن این اسم هنگامی که میخواهیم این ایمیج را با دستور docker run اجرا کنیم سخت و مشکل است. برای رفع این مشکل می توانیم از قابلیت نامگذاری داکر استفاده کنیم و یک اسم برای این ایمیج خود بگذاریم. برای این کار باید مطابق دستور زیر عمل کنیم.

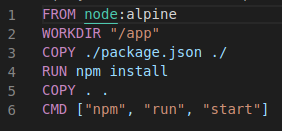
Docker build -t mpouaykh/redis:v2 **.**

ابتدا دستور t- را میگذاریم و سپس Docker ID خود را مشخص میکنیم و بعد اسم سرویس خود و نسخه ای که مد نظر ما است را مشخص میکنیم. همچنین با استفاده از دستور push میتوانیم این ایمیج را در داکر هاب بارگذاری کنیم.

Docker push mpouyakh/redis:v2

بعد از توضیحاتی که در مورد نحوه ایمیج ساختن آرایه کردیم؛ سراغ نوشتن Dockerfile برای سرویس های خود میرویم. Dockerfile برای این سرویس ها کمی متفاوت است ولی ساختار اصلی همان سه بخش اصلی است.

حال در این بخش ما میخواهیم سرویس های client که با استفاده از react نوشته است و سرویس های server و worker که با استفاده از node js نوشته شده اند را ایمیج کنیم. برای نوشتن dockerfile این سرویس باید تغییراتی نسبت به dockerfile قسمت قبل بدهیم. در شکل زیر dockerfile برای این سرویس ها را بررسی میکنیم.



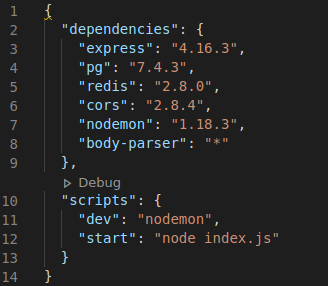
شکل۶ - dockerfile سرویس های server؛ worker و client

همان طور که مشاهده میکنیم؛ این dockerfile کمی متفاوت تر از dockerfile قسمت قبلی است. ایمیج پایه ما node است به دلیل آنکه سرویس های ما با node js و react js نوشته شده اند و برای آنکه ما پکیج منیجر npm را داشته باشیم باید از این ایمیج پایه استفاده کنیم. نسخه alpine برای این منظور است که فقط برنامه های پایه مانند node و npm در این ایمیج وجود دارد و برنامه های مختلف و جانبی دیگر وجود ندارد. ما با استفاده از npm می توانیم برنامه ها و وابستگی هایی که سرویس های ما نیاز دارند را در کانتینر دانلود و نصب کنیم. با استفاده از ایمیج پایه قبلی؛ داکر سرور دچار مشکل میشد به دلیل آنکه در ایمیج پایه قبلی npm نصب نبوده است. در خط دوم ما مشخص میکنیم که برنامه ها و وابستگی هایی که دانلود می شوند ؛ در کدام مسیر در داخل کانتینر قرار بگیرند. در اینجا ما مسیر app/ را انتخاب کردیم. در خط سوم فایل package.json شامل وابستگی هایی که برنامه ما دارد و باید آن وابستگی ها را در کانتینر نصب کنیم تا برنامه ما درست کار کند و همچنین یک بخش دیگر برای دستوراتی که موقع اجرای کانتنیر میتوانیم استفاده کنیم را شامل میشود. در شکل زیر package.json مربوط به سرویس server را میتوانیم مشاهده کنیم. با دستور COPY ما این فایل را در مسیری که در خط ۲ مشخص کردیم کپی میکنیم.

"./" اسلش نقطه به این معنی است که در همان مسیری که dockerfile ما وجود دارد یا مسیری که در خط ۲ مشخص کردیم.

در خط ۴ وابستگی هایی که در فایل package.json مشخص کردیم را دانلود و نصب میکند و در خط ۵ همه فایل های مسیر فعلی را در مسیر خط ۲ کپی میکند. در آخر هم با استفاده از دستور خط ۶سرویس را اجرا میکند.

نکته ای در این dockerfile وجود دارد این است که ما دوبار از دستور copy استفاده کردیم؛ یکی در خط ۳ و ۵. میتوانستیم فقط از دستور خط ۵ قبل از خط چهار استفاده کنیم. در این صورت هر بار که ما تغییری در کد میدادیم؛ همه ی برنامه ها و وابستگی ها دوباره دانلود و نصب میشدند. حال برای آنکه دوباره کاری انجام نشود هر سری که ما کد خود را تغییر میدهیم؛ باید package.json را جدا کنیم و قبل از دستور RUN قرار بدهیم و بقیه فایل های دیگر را بعد از دستور RUN قرار میدهیم. این کار باعث میشود تا هر بار کد خود را تغییر میدهیم؛ لازم نباشد ما همه ی وابستگی ها دوباره دانلود و نصب کنیم.



شکل ۷ - فایل package.json برای سرویس server

با استفاده از این فایل می توانیم وابستگی ها و برنامه هایی که سرویس ما نیاز دارد را مشخص کنیم و با استفاده از دستور"RUN npm install" این ها را نصب کنیم. همانطور که در شکل معلوم است برنامه های redis و postgres در این بخش آمده اند که این سرویس با استفاده از این برنامه ها به سرور redis و postgres وصل میشود. وابستگی دیگر فریم ورک express است که ما برای اجرا شدن برنامه خود نیاز داریم. قسمت script دستوراتی که میتوانیم در کانتینر خود اجرا کنیم آورده میشود. دستور اول زمانی استفاده میشود که ما میخواهیم تغییرات زیادی در کد خود بدهیم و نمیخواهیم هر بار برای تغییرات جدید ایمیج جدید درست کنیم. وقتی از این دستور استفاده میشود تغییرات به صورت آنی در کانتینر ما ظاهر میشود به شرطی که یک سری تنظیمات را در فایل docker-compose اعمال کنیم. قسمت دوم که "start" است برای اجرا شدن برنامه node js ما استفاده می شود که در dockerfile سرویس server از این بخش استفاده کردیم. Package.json برای سرویس worker و client هم به همین صورت است فقط با کمی تفاوت در وابستگی ها.

حال ایمیج های خود را با دستور "docker images" میتوانیم در شکل زیر برای ۳ سرویس خود مشاهده کنیم.



شکل ۸ - ایمیج های ساخته شده برای سرویس های worker؛ server و client که در اینجا به اسم my-app است

هر کدام از ایمیج های ما ID و اسم مخصوص به خود را گرفته اند که میتوانیم این سرویس ها را بر اساس نامشان اجرا کنیم.

۳ جمع بندی:

در این فصل، ما با مایکرو سرویس هایی که قرار است در پلتفرم کوبرنتیز بارگذاری کنیم و با استفاده از این پلتفرم آنها را مدیریت و اجرا کنیم، آشنا شدیم. اجزا و نحوه ارتباطات این مایکرو سرویس ها با یکدیگر را شناختیم. در آخر هم نحوه ایمیج کردن این مایکرو سرویس ها با استفاده از سیستم داکر توضیح داده شد. بدون این فصل امکان استفاده از پلتفرم کوبرنتیز ممکن نبود و این فصل بخش مهمی برای آماده سازی استفاده از پلتفرم کوبرنتیز به شمار می رود.

فصل چهارم : راه اندازی پلتفرم کوبرنتیز و نوشتن کانفیگ فایل های آبجکت های کوبرنتیز

بعد از آنکه سرویس های خود را نوشتیم و ایمیج کردیم، حال سراغ راه اندازی کلاستر خود می رویم و پلتفرم کوبرنتیز را به صورت محلی (local) راه اندازی میکنیم. بعد از راه اندازی، باید آبجکت های مختلف در کوبرنتیز را درست کنیم که بتوانیم این سرویس ها را به صورت خودکار مدیریت کنیم. در این فصل آبجکت های مختلف پلتفرم کوبرنتیز را معرفی و توضیح می دهیم و در آخر کانفیگ فایل های این آبجکت ها را می نویسیم و در پلتفرم کوبرنتیز بارگذاری میکنیم تا این پلتفرم بتواند سرویس های ما را شناسایی کند و آنها را مدیریت کند‌.

۱ راه اندازی کلاستر کوبرنتیز

در این پروژه ما از یک گره رهبر و دو گره کارگر استفاده میکنیم. یکی از گره ها یک ماشین فیزیکی مجزا با ۳ هسته cpu و ۴ گیگ رم است. گره کارگر بعدی یک ماشین مجازی در ماشین فیزیکی است با ۲ هسته cpu و ۴ گیگ رم. سیستم عامل ۳ گره Ubuntu 19.10 است. برای اتصال این گره ها به یکدیگر؛ ما از kubeadm استفاده میکنیم.

برای راه اندازی کلاستر خود باید مراحل زیر را اجرا کنیم.

الف) باید داکر بر روی همه گره ها نصب شود با استفاده از دستور زیر این کار را انجام میدهیم.

sudo apt-get update \

&& sudo apt-get install -qy docker.io

ب) حال repository های مربوط به پکیج های کوبرنتیز را با استفاده از دستورات زیر به سیستم خود اضافه میکنیم.

sudo apt-get update \

&& sudo apt-get install -y apt-transport-https \

&& curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key add -

echo "deb http://apt.kubernetes.io/ kubernetes-xenial main" \

| sudo tee -a /etc/apt/sources.list.d/kubernetes.list \

&& sudo apt-get update

ج) حال باید پکیج های مختلف مربوط به کلاستر کوبرنتیز را نصب کنیم.

sudo apt-get update \

&& sudo apt-get install -yq \

kubelet \

kubeadm \

Kubernetes-cni

Kubernetes-cni مربوط به بخش های نتورکینگ در کوبرنتیز است. cni مخفف container Network Interface است که مسیول ارتباطات کانتینر ها با network driver ها هستند. Container runtime مسئولیت تخصیص دادن IP به کانتینر ها را به دوش cni میگذارد و cni به network driver متصل میشود و یک IP برای کانتینر میگیرد.

د) باید پکیج های خود را در حالت on hold قرار بدهیم که وقتی اپدیت جدید آمد؛ کلاستر ما بهم نریزد.

sudo apt-mark hold kubelet kubeadm kubectl

ه) swap را باید در سیستم خود خاموش کنیم چون ممکن است swap memory باعث خطا های غیر قابل پیش بینی شود. برای این منظور میتوانیم از دستور sudo swapoff -a یا اینکه در مسیر etc/fstab/ بخش های مربوط به swap را کامنت کنیم.

ر) باید در مسیر /etc/hosts نام ها و IP های گره های مختلف را برای هر گره وارد کنیم تا این گره ها بتوانند همدیگر را ping کنند.

ف) حال باید دستور زیر را در گره رهبر وارد کنیم.

Kubeadm init --pod-network-cidr=10.244.0.0/16 --apiserver-advertise-address=192.168.43.159

در بخش اول این دستور ما subnet پاد های خود را مشخص میکنیم و در بخش دوم api server خود را با استفاده از IP ماشین خود به بقیه نود ها تبلیغ می کنیم تا بتوانند به گره رهبر متصل شوند. بعد از این مرحله؛ باید دستورات زیر را وارد کنیم تا بتوانیم به کلاستر دسترسی پیدا کنیم.

Mkdir -p $HOME/.kube

Sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config

sudo chown $(id -u):$(id -g) $HOME/.kube/config

حال ما با استفاده از flannel نتورکینگ بین پاد ها را تنظیم میکنیم و با استفاده از دستور زیر در کلاستر خود این pod network را راه اندازی میکنیم.

export ARCH=amd64

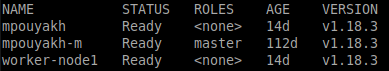
curl -sSL "https://github.com/coreos/flannel/blob/master/Documentation/kube-flannel.yml?raw=true" | sed "s/amd64/${ARCH}/g" | kubectl create -f -

م) دستور زیر را در گره های کارگر وارد می کنیم تا بتوانیم به گره رهبر متصل بشویم.

kubeadm join 192.168.43.159:6443 --token 0gvohk.nvz098qjy0ymupao \

--discovery-token-ca-cert-hash sha256:0561d8a5a0c56a3564303c36075275cb66724bc72a31af3b6dc94d561b2f52c6

بعد از انجام این مراحل میتوانیم وضعیت گره های خود را با استفاده از دستور"kubectl get nodes" ما باید به صورت زیر باشند.



شکل ۱ - وضعیت گره ها بعد از راه اندازی کلاستر

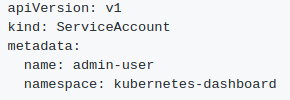
همچنین ما میتوانیم با استفاده از داشبورد کوبرنتیز که یک web UI است به کلاستر خود دسترسی پیدا کنیم و راحت تر بتوانیم کلاستر خود را مدیریت کنیم.

برای راه اندازی این داشبورد مراحل زیر را باید انجام دهیم.

الف) ابتدا باید دستور زیر را اجرا کنیم تا قسمت های مختلف این داشبورد در کلاستر نصب شوند.

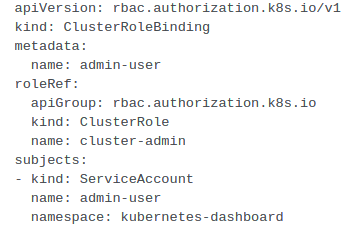
kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/v2.0.0/aio/deploy/recommended.yaml

ب) با استفاده از آبجکت های service account و ClusterRoleBinding که در فصل بعدی بیشتر در مورد این آبجکت توضیح خواهیم داد یک دسترسی admin به کاربر جدید می دهیم که بتواند به همه منابع و namespace ها دسترسی پیدا کند.



شکل ۲ - آبجکت سرویس اکانت برای کاربر جدید

با استفاده از این آبجکت کوبرنتیز ما کاربر خود را مشخص میکنیم و همچنین namescpace این آبجکت هم معین میشود.



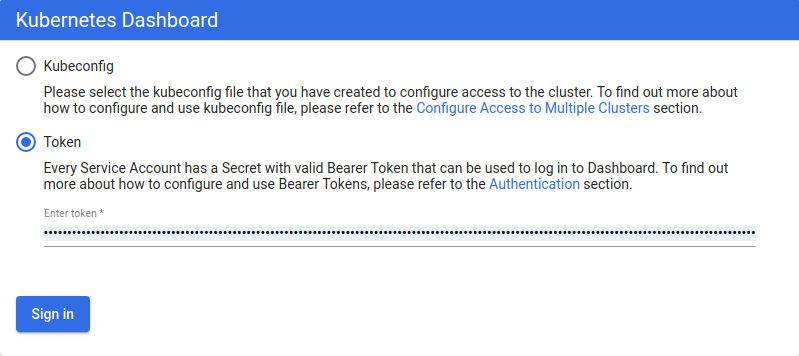
شکل ۳- اتصال کاربر جدید به نقش ادمین کلاستر

با استفاده از این آبجکت کوبرنتیز ما می توانیم به کاربر جدید دسترسی ادمین را بدهیم. به دلیل آنکه نقش ادمین قبلا تعریف شده است دیگر لازم نیست ClusterRole را تعریف کنیم و نتها با استفاده از این فایل کاربر جدید نقش ادمین میگیرد.

برای ورود به این داشبورد باید یک token دریافت کنیم که با استفاده از دستور زیر میتوانیم این token را تولید کنیم.

kubectl -n kubernetes-dashboard describe secret $(kubectl -n kubernetes-dashboard get secret | grep admin-user | awk '{print $1}')

سپس با استفاده از این token وارد داشبورد کوبرنتیز می شویم.



شکل ۴- وارد کردن token مربوط به داشبورد کوبرنتیزبرای وارد شدن

2 معرفی آبجکت های مهم کوبرنتیز و نوشتن کانفیگ فایل های بعضی از این آبجکت ها:

2-1 (RBAC) Role-Based Access Control

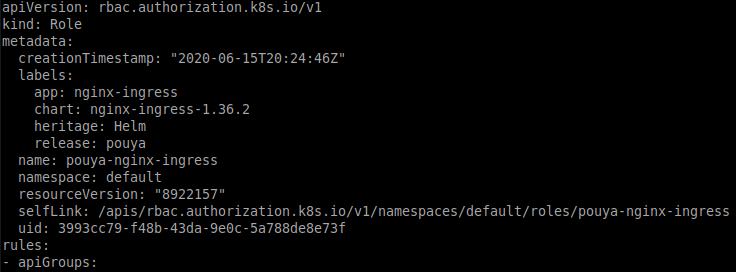
در کلاستر ما دو نوع کاربر داریم: یک کاربر حقیقی (User Accounts)که می تواند به کلاستر ما دسترسی پیدا کند و منابع را تغییر یا ایجاد کند و یک کاربر دیگر (Service Accounts) که برنامه ها و پروسه های درون کلاستر ما هستند که آنها هم میتوانند تغییراتی در کلاستر ما ایجاد کنند. به منظور

آنکه ما این دسترسی ها را برای این کاربر ها یا برنامه ها محدود کنیم از این قابلیت کوبرنتیز استفاده میکنیم. Service Account یک آبجکت کوبرنتیز است برای برنامه ها و پروسس ها معمولا توسط api server به صورت خودکار ایجاد میشود. به طور مثال در فصل قبلی ما برای آنکه داشبورد کوبرنتیز را راه بیندازیم یک service account نوشتیم که مشخص میکند که نام این سرویس چه باشد و در کدام namespace قرار بگیرد.

برای آنکه ما بتوانیم با api-server ارتباط برقرار کنیم باید احراز هویت شویم. این احراز هویت برای کاربران حقیقی جدید به صورت client- certificate است. این نوع احراز هویت بر اساس کلید های عمومی و خصوصی انجام میشود و باید certificate authority ای وجود داشته باشد که درخواست احراز هویت کاربر جدید را امضا کند تا اجازه پیدا کند به کلاستر دسترسی پیدا کند. برای مثال میخواهیم یک کاربر جدید را اضافه کنیم. ابتدا باید با استفاده از ابزار openssl یک کلید خصوصی و certificate بگیرد و سپس یک certificate signing request با همین ابزار بسازد و این درخواست را به صورت فایل Yaml درآورد. بعد از آنکه این درخواست توسط CA تایید شد حال کاربر جدید به کلاستر ما اضافه میشود.

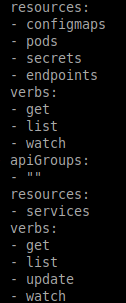
برای احراز هویت درخواست های سرویس ها هم باید از service account tokens استفاده کنیم که برای هر سرویس یک token به صورت آبجکت secret که باعث میشوند token های ما encode شوند و اطلاعات ما امن تر باشد؛ به این سرویس اختصاص می یابد. با استفاده از این token ها میتوانند با api server ارتباط برقرار بکنند.

بعد از مرحله احراز هویت ما میتوانیم با استفاده از آبجکت های Role و ClusterRole دسترسی های کاربران و سرویس ها را محدود کنیم و فقط بتوانند به بخشی از منابع دسترسی پیدا کنند و بعضی از عملیات ها را انجام دهند. آبجکت Role مشخص میکند که در یک namespace مشخص چه عملیاتی میتوانند انجام دهند و به چه منابعی می توانند دسترسی پیدا کنند ولی آبجکت ClusterRole مشخص میکند که در سطح کلاستر چه عملیاتی میتواند انجام دهد و به چه منابعی میتواند دسترسی پیدا کند.



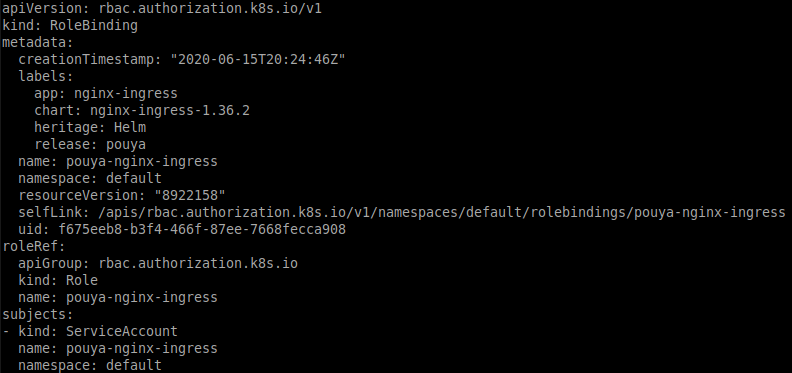
شکل ۵- آبجکت Role برای سرویس nginx

همانطور که مشاهده می کنیم؛ آبجکت Role برای سرویس ما نوشته شده است و مشخص شده است که این سرویس به کدام namespace دسترسی داشته باشد. در شکل زیر هم عملیات هایی را که برای برخی از منابع میتواند انجام دهد در شکل زیر مشاهده می کنیم.



شکل ۶- عملیات هایی(verbs) که سرویس ما بر روی برخی از منابع میتواند انجام دهد

آبجکت Role به تنهایی نمی تواند کاری انجام دهد. برای آنکه این آبجکت کار کند باید از آبجکت RoleBinding استفاده کنیم که service account یا کاربران را به namespace ای که مشخص کردیم اتصال بدهد.



شکل ۷- آبجکت RoleBinding برای وصل کردن کاربران یا service account ها به آبجکت Role

آبجکت های ClusterRole و ClusterRoleBinding به همین شکل است فقط با این تفاوت که این آبجکت ها در سطح کلاستر عمل میکنند.

2-2 پاد ها(Pods) ؛ رپلیکا ست ها(Replica sets) و دپلویمنت ها (Deployments)

ساده‎ترین واحدی که در Kubernetes وجود دارد Pod ها هستند. در واقع Container به خود ماشین (سرور) تعلق ندارد، بلکه یک یا چند Container در یک Pod قرار دارند. در دید کلی یک Pod شامل یک یا چند Container است که به عنوان یک اپلیکیشن یا سرویس شناخته می‎شوند. Pod ها شامل Container هایی هستند که با یکدیگر کار می‎کنند و یک چرخه‎ی حیات دارند. همچنین همیشه آن‌ها باید در یک Node قرار بگیرند. این کانتینر ها به یک حافظه داخلی یا خارجی مشخص دسترسی دارند و در یک فضای ارتباطی(Network namespace) قرار دارند.

Pod ها به عنوان یک واحد، مدیریت می‎شوند و فضا، منابع و IP را با هم به اشتراک می‎گذارند. معمولا Pod ها شامل یک یا چند Container هستند که هدفشان این است که یک وظیفه را به خوبی انجام دهند. سرویس و برنامه‎ها زمانی کار می‎کنند که Pod ها در حال اجرا باشند. به عنوان مثال در یک Pod ممکن است یک Container سرویس اصلی را اجرا کند و Container دیگر در Database تغییرات را اعمال و همچنین زمانی که منابع خارجی متصل می‎شوند، آن را شناسایی می‎کند.

هنگامی که با Kubernetes کار می‎کنیم به جای مدیریت مستقیم و تکی Pod ها، ما گروه‎هایی از Podها را مدیریت می‎کنیم که شامل Pod های کپی شده هستند. ما از قالب آماده‎ی Pod ها استفاده می‎کنیم و Pod ها را به صورت افقی در Nodeهای کلاستر توزیع می‎کنیم (کپی می‎کنیم) که اینکار توسط deployment و Replication Set انجام می‎شود.

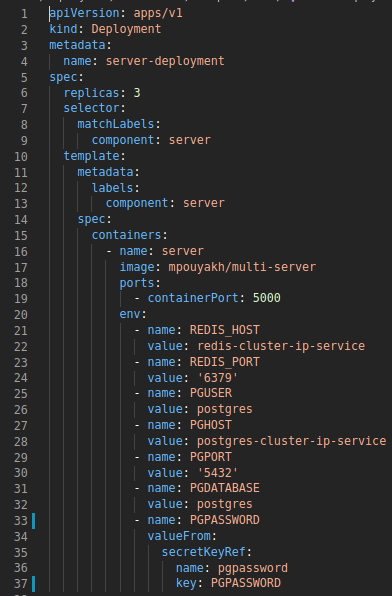
Deployment یک شیء سطح بالاست که به منظور تسهیل مدیریت چرخه‎ی حیات برای Podهای کپی شده در کلاستر طراحی شده است. پیکربندی Deployment را به راحتی می‎توان تغییر داد و تعداد Replica Set ها را تنظیم کرد. به دلیل وجود این ویژگی‏‎ها Deployment ابزار بسیار پرکاربردی است که در Kubernetes با آن اغلب کار می‎کنید.

deployment ابزاری برای تعریف Pod است که کپی، توزیع یکسان، افزایش و کاهش Podهای در حال اجرا را بر عهده دارد. این یک راه آسان برای Load Balancing، افزایش و کاهش Pod ها در صورت افزایش حجم کاری در Kubernetes است. deployment می‎داند که چگونه Podهای مورد نیاز را ایجاد کند؛ زیرا در پیکربندی خود یک template کپی شده از یک Pod دارد.

deployment مطمئن می‎شود که تعداد Pod هایی که در Node باید قرار بگیرد با تعداد Podهایی که در تنظیمات خودش است منطبق باشد. اگر Host یا Pod های در آن از بین برود، جهت جبران، Controller شروع به ساختن Pod های جایگزین می‎کند. اگر تعداد کپی‎ها در پیکربندی Controller تغییر کند، Controller بلافاصله شروع به اعمال تغییرات می‎کند (تعداد Pod های کپی شده را تغییر میدهد) مانند اضافه یا حذف کردن Pod ها. deployment همچنین می‎تواند Pod ها را نیز به نسخه‎ی جدید خود بروزرسانی کند یا اگر نسخه جدید عملکرد خوبی نداشت به نسخه قدیمی برگرداند.

Replication Set برخلاف deployment قابلیت Rollback و Rolling Update، چرخه‌ی بروزرسانی Pod ها را ندارد، در عوض deployment در سطح بالاتر این ویژگی‌ها را فراهم می‎کند. پس ما نیازی به کنترل replication set نداریم و خود deployment آن را ایجاد و مدیریت میکند.

حال آبجکت deployment یکی از سرویس های خود را به اسم سرور با هم بررسی میکنیم.



شکل ۸ - آبجکت deployment برای سرویس server

در این آبجکت ابتدا مشخص میکنیم که به کدام api در api server متصل شویم و این آبجکت را بسازیم. در خط دوم مشخص میکنیم که نوع آبجکت ما چه است؛ این نوع می تواند service که در بخش بعدی توضیح خواهیم داد و مربوط به شناسایی پاد ها در کلاستر است؛ یا میتواند pod باشد و یا آبجکت های دیگر. در قسمت spec اولی مشخص میکنیم که تعداد پاد هایی که میخواهیم در کلاستر ما اجرا شود چه تعداد باشد و همچنین label برای این سرویس ما مشخص میکند تا بتوان به جای اسم یا ID این آبجکت از این label برای گروه بندی این پاد ها در کلاستر استفاده شود. پاد های ما با استفاده از قسمت template تولید میشوند. در این قسمت ایمیج مربوط به سرویس ما مشخص شده است که پاد ما از این ایمیج درست میشود. همچنین سرویس ما بر روی پورت ۵۰۰۰ در داخل کانتینر اجرا میشود. در بخش آخر ما یک سری Environment variable هایی را باید ست کنیم تا بتوانیم به پایگاه داده های redis و postgres وصل شود.این variable ها در فایل keys.js که در کانتینر ما این فایل وجود دارد؛ آمده است تا بتوانیم به این پایگاه داده ها متصل شویم. همان طور که در فصل قبل مشاهده کردیم سرویس server ما به این دو پایگاه داده متصل است. همچنین سرویس worker ما هم به پایگاه داده redis متصل است که بخش اتصال به redis را در آبجکت deployment خود دارد. پورت این سرویس های پایگاه داده مشخص شده است و قسمت value مشخص میکند که این پایگاه داده ها در کدام پاد ها اجرا میشوند و سرویس ما بتواند به این سرویس های پایگاه داده متصل شود. بخش نتورکینگ پاد ها در بخش بعدی بیشتر توضیح داده خواهد شد. برای اینکه بتوانیم به پایگاه داده postgres متصل شویم باید یک رمز ایجاد کنیم و با استفاده از این رمز به این پایگاه داده متصل شویم. برای آنکه این پسورد در معرض دید نباشد از آبجکت secret استفاده میکنیم. دستور ایجاد این آبجکت و گذاشتن پسورد در این آبجکت به شکل زیر است.

Kubectl create secret generic pgpassword --form-literal PGPASSWORD=12345678

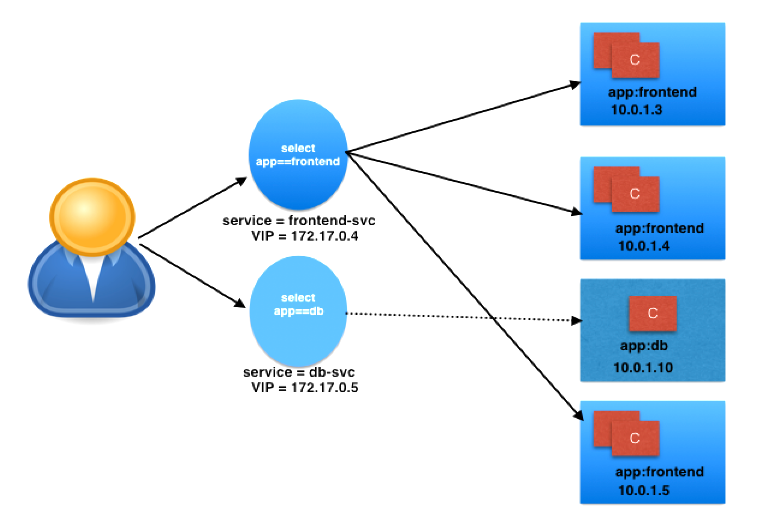
بخش form -literal-- به این منظور است که ما این آبجکت را با استفاده از دستور ترمینال اجرا میکنیم و نه به صورت نوشتن فایل و بعد بارگذاری آن در کوبرنتیز.

2-3 Service ها

در Service ،Kubernetes جزئی است که شبکه سازی بین پاد ها را ایجاد میکند و ترافیک را به Pod مورد نظر هدایت می‎کند. یک سرویس شامل گروه‎هایی منطقی از Pod ها هستند که با هم یک عملکرد را دارند. به عنوان مثال: Web Service، Mail Service

Service به شما اجازه می‎دهد تا مسیر مناسب برای رسیدن به Container مناسب فراهم گردد. در عین حال Service به شما اجازه‎ی گسترش یا جایگزینی را در Backend در صورت نیاز می‎دهد. یک Service بدون در نظر گرفتن تغییرات در Podها مسیریابی آن را انجام می‎دهد. با اجرا و عملیاتی سازی Service، شما به راحتی می‎توانید ساختار Container ها را تغییر دهید.

همانطور که میدانیم وقتی پاد ها ایجاد میشوند؛ یک IP به آنها تخصیص میابد و اگر بخواهیم بر اساس این IP ها به پاد ها دچار مشکل میشویم چون پاد ها همیشه یک آدرس مشخص ندارند و ممکن است از بین بروند و یک آدرس جدید بگیرند. برای حل این مشکل ما از آبجکت Service استفاده میکنیم که با استفاده از label هایی که در deployment استفاده شد؛ این پاد ها را گروه بندی میکند و ترافیک را به این پاد ها میفرستد. با این تکنیک دیگر لازم نیست نگران تغییر آدرس های پاد ها باشیم.

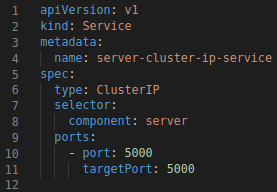


شکل ۹ - نحوه عملکرد آبجکت service در پیدا کردن پاد ها

همانطور که در شکل بالا مشاهده میکنیم؛ هر کدام از پاد ها یک آدرس مخصوص به خود دارند. ولی آبجکت service این پاد ها را بر اساس label مدیریت میکند و ترافیک را به این پاد ها میفرستد. همچنین هر یک از این آبجکت های service یک آدرس مخصوص (clusterIP)به خود میگیرند که بر اساس این آدرس؛ پاد های دیگر میتوانند به پاد های دیگر از طریق این آبجکت دسترسی پیدا کنند. این نوع آبجکت ClusterIP نام دارد که فقط به صورت محلی قابل دسترسی است و ترافیکی از بیرون کلاستر به این پاد ها نمیتوان فرستاد. آدرس های VIP همان آدرس های ClusterIP برای دو اپلیکیشن db و frontend هستند.

این clusterIP ها و آدرس های پاد ها در جدولی به نام iptables قرار دارند که توسط kube-proxy که در هر گره وجود دارد؛ مدیریت میشود و این آدرس ها را اضافه یا حذف میکند.

حال آبجکت service برای سرویس server خود را مشاهده میکنیم.

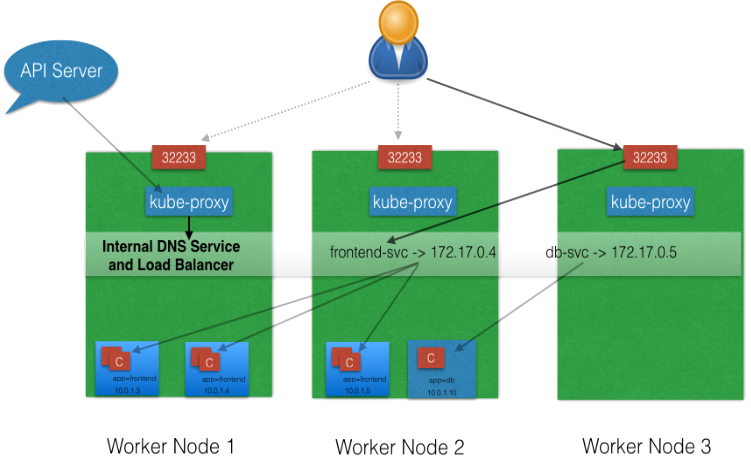


شکل ۱۰ - آبجکت service برای سرویس server

در این شکل می توان ساختار آبجکت service را مشاهده کرد. در خط ۶ ام ما نوع این service را مشخص کردیم که این نوع می تواند مقادیر همچون NodePort و LoadBalancer باشد. در بخش بعدی بیشتر در مورد این دو مورد صحبت خواهیم کرد. در خط ۸؛ میبینیم همان label ای که در آبجکت deployment برای این سرویس مشخص کرده ایم؛ آمده است و آبجکت service این پاد ها را بر اساس label مدیریت میکند و نه بر اساس آدرس پاد ها. پورتی که این سرویس را می خواهیم در معرض کاربران برای دسترسی قرار دهیم؛ در خط ۱۱ آمده است و targetPort نام دارد. Port در خط ۱۰ برای این منظور است که اگر پاد های دیگر قصد این را داشتند به این پاد دسترسی داشته باشند از طریق این پورت دسترسی پیدا کنند. این پورت میتواند از targetport متفاوت باشد. این آبجکت سرویس برای بقیه سرویس های این پروژه به همین صورت است و تفاوت آنچنانی نمیکند.

2-3-1 Nodeport و LoadBalancer

به منظور آنکه بتوانیم خارج از کلاستر به پاد های داخل کلاستر دسترسی پیدا کنیم؛ میتوانیم از NodePort استفاده کنیم. با انتخاب پورتی از بازه ۳۰۰۰ تا ۳۲۷۶۷؛ میتوانیم از طریق هریک از گره های داخل کلاستر به پاد های خود دسترسی پیدا کنیم. در شکل زیر به صورت ملموس تر میتوانیم مشاهده کنیم که کاربر با پورت ۳۲۲۳۳ توانسته از طریق یکی از گره ها به اپلیکیشن مد نظر خود دسترسی پیدا کند. این پورت به ClusterIP سرویس اپلیکیشن متصل شده است. وقتی کاربر با پورت مشخص به اپلیکیشن میخواهد دسترسی پیدا کند؛ این دسترسی از طریق service این اپلیکشن است و آبجکت service است که ترافیک را به این پاد ها میفرستد.



شکل ۱۱ - نحوه وصل شدن کاربر از خارج کلاستر به اپلیکیشن با استفاده از NodePort

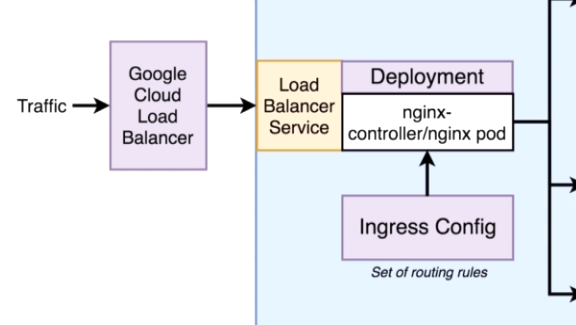
این نوع دسترسی از خارج کلاستر خیلی مطلوب نیست و بیشتر برای تست استفاده میشود. روش مناسب تری که میتوانیم برای آنکه از خارج کلاستر به پاد ها دسترسی پیدا کنیم؛ استفاده کنیم LoadBalancer است. این Loadbalancer به صورت خودکار ClusterIP و NodePort را برای اپلیکشین ما راه اندازی میکند و ترافیک های خارجی را از طریق یک پورت مشخص به اپلیکیشن ما میفرستد. دو مشکل در این روش وجود دارد : ۱. برای استفاده از این LoadBalancer باید حتما کلاستر خود را در یکی از فراهم کننده های ابر مانند Google یا AWS راه اندازی کرده باشید تا بتوانید از این قابلیت استفاده کنید. ۲. به دلیل آنکه این LoadBalancer ها بخش های سخت افزاری هستند و یک کارت شبکه جدا و IP جدا دارند؛ هزینه زیادی را برای ما تحمیل میکنند اگر بخواهیم برای هر یک از اپلیکیشن های خود از این LoadBalancer استفاده کنیم.

برای حل این دو مشکل میتوانیم از نوع ۴ ام Service استفاده کنیم که Ingress نام دارد.

2-3-2 Ingress

با استفاده از این آبجکت؛ ما میتوانیم اپلیکیشن های خود را به محیط بیرون دسترس پذیر کنیم و هنگامی که درخواست ها وارد کلاستر ما میشوند از طریق Ingress به اپلیکیشن های مختلف میفرستد. حال با Ingress لازم نیست ما برای هر اپلیکیشن خود یک LoadBalancer اختصاص دهیم و تنها میتوانیم این LoadBalancer را به Ingress متصل کنیم و این آبجکت کار Load Balancing را در داخل کلاستر ما انجام دهد.

با استفاده از آبجکت Ingress ما میتوانیم فقط Rule های مسیریابی را مشخص کنیم و بگوییم هر درخواست به کدام اپلیکیشن باید فرستاده شود. ولی برای آنکه بتوان این مسیریابی را انجام داد باید از یک کنترلر مانند deployment؛ به نام Ingress-controller استفاده کنیم که آبجکت Ingress را دریافت میکند و یک پاد درست کند و کار LoadBalancing را انجام میدهد. پروژه Ingress-Nginx یک پروژه آماده است که این Ingress-controller را برای ما فراهم میکند.

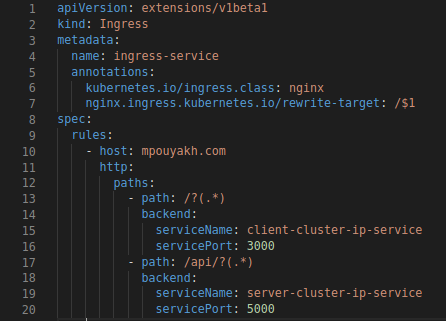


شکل ۱۲ - نحوه کارکرد سرویس LoadBalancer با Ingress

نحوه کارکرد کلی این دو نوع service با هم را در شکل بالا مشاهده میکنیم. LoadBalancer فراهم کننده ابر به یک سرویس LoadBalancer در داخل کلاستر ما متصل میشود و این Service هم ترافیک را به پاد Nginx میفرستد که با استفاده از Nginx controller ما مسیر هایی که این پاد باید بفرستد را مشخص کردیم.

همان طور که در شکل بالا مشخص است یک کانفیگ فایل مربوط به آبجکت Ingress است که ما به منترلر میدهیم تا این مسیر ها را در پاد پیاده سازی کند و Nginx pod بتواند درخواست ها را به سرویس های مختلف بفرستد.

حال کانفیگ فایل Ingress برای این پروژه را در شکل زیر مشاهده می کنیم.



شکل ۱۳ - کانفیگ فایل آبجکت Ingress

مانند همه کانفیگ فایل ها باید api مورد نظر و نوع آبجکت را مشخص کنیم. در خط ۶ ما به کوبرنتیز می گوییم که میخواهیم از پروژه nginx برای مسیریابی در کلاستر خود استفاده کنیم. در خط ۷ مشخص میکنیم که بعد از آنکه مسیر مشخص شد؛" api/" به "/" تغییر پیدا کند چون ما در server مسیر هایی که مشخص کرده ایم api/ ندارند. در بخش spec هم اپلیکیشن هایی که میخواهیم درخواست ها را بفرستیم؛ معین کرده ایم. در بخش host یک domain name به صورت محلی تعریف کرده ایم که باید بعدا آدرس ClusterIP برای Ingress را در etc/hosts/؛ برای این domain name بگذاریم. همان طور که معلوم است دو مسیر "/" و " api/" وجود دارد که اولی به ClusterIP سرویس client اشاره میکند و دومی به ClusterIP سرویس server اشاره میکند. در نتیجه درخواست ها به این گونه به این دو سرویس فرستاده میشود.

برای نصب پروژه Ingress-Nginx میتوانیم از پکیج منیجر Helm استفاده کنیم. این پکیج منیجر به کلاستر ما دسترسی پیدا میکند و برنامه های مختلف را در کلاستر نصب میکند.

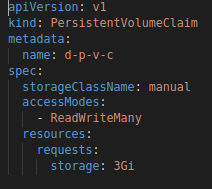
helm install pouya stable/nginx-ingress

با استفاده از این دستور؛ Helm همه بخش های مختلف Nginx را در کلاستر راه اندازی میکند.

2-4 Persistent Volumes

کانتینر ها وقتی اجرا می شوند؛ یک قسمتی از حافظه ماشین ما به این کانتینر اختصاص میابد. زمانی که این کانتینر در حال اجرا شدن است؛ داده هایی که تولید می شوند در فایل سیستم مخصوص این کانتینر قرار میگیرد. حال اگر کانتینر ما از بین رفت یا دوباره شروع به کار کرد آیا این داده هایی که توسط کانتینر تولید شده بود باقی میماند؟ جواب خیر است. همه داده های کانتینر پاک میشود و همه چیز از نو شروع میشود. همچنین حافظه ای که در داخل پاد است هم به همین صورت است و با از بین رفتن پاد همه داده ها پاک میشود. حال برای آنکه بتوانیم این داده ها را نگهداری کنیم تا از بین نروند باید از حافظه های مخصوص که خارج از این کانتینر ها و پاد ها قرار دارند استفاده کنیم. نام این آبجکت در کوبرنتیز persistent volume نام دارد که حافظه مورد نیاز برای یک کانتینر را فراهم میکند.

برای آنکه ما بتوانیم این حافظه های پایدار را بدست بیاوریم باید از یک آبجکت دیگر به اسم Persisitnet Volume Claim استفاده کنیم که با استفاده از این آبجکت میزان حجم حافظه ای را که نیاز داریم از کوبرنتیز درخواست میکنیم و کوبرنتیز بر اساس منابعی که در اختیار دارد این حافظه را به پاد مورد نظر تخصیص میدهد.



شکل ۱۴ - آبجکت PersistentVolumeClaim

با استفاده از این درخواست ما ۳ گیگ را از کوبرنتیز درخواست میکنیم تا به پاد مد نظر ما اختصاص دهد. دسترسی برای این بخش حافظه میتواند به سه بخش تقسیم شود.

الف) ReadWriteOnce : این دسترسی فقط اجازه میدهد در لحظه یک کاربر به این حافظه دسترسی داشته باشد و عملیات نوشتن و خواندن را انجام دهد.

ب) ReadOnlyMany : با استفاده از این دسترسی چند کاربر همزمان میتوانند از این حافظه فقط بخوانند.

ج) ReadWriteMany: این دسترسی اجازه میدهد که چند کاربر در لحظه بتوانند در این حافظه عملیات خواندن و نوشتن را انجام دهند.

پس کوبرنتیز حافظه ای که ما نوع دسترسی آن را مشخص میکنیم برای پاد مد نظر ما فراهم میکند.

نکته ای دیگری که در این کانفیگ فایل وجود دارد؛ قسمت StorageClass است که تعیین میکند که آیا فراهم کردن این حافظه به صورت static انجام میشود یا dynamic.

Static به این صورت است که خود ادمین باید آبجکت Persistent Volume را بسازد و به api server بدهد . به دلیل آنکه در این قسمت manual آمده است؛ این به این منظور است که باید توسط خود ادمین این آبجکت تولید شود. برای اینکه ما به صورت خودکار این آبجکت Persistent volume را بسازیم؛ میتوانیم از NFS-provisioning استفاده کنیم.

2-4-1Dynamic NFS Provisioning

NFS-client provisioner یک فراهم کننده حافظه به صورت خودکار است. که ما این فراهم کننده را در کلاستر خود با استفاده از پکیج منیجر Helm که بخش های مختلف مانند RBAC ؛ Deployment و StorageClass را در کوبرنتیز به طور خودکار ایجاد می کند . این فراهم کننده یک پاد در داخل کلاستر ما است که به آبجکت Storage Class متصل میشود. این فراهم کننده درخواست هایی که برای حافظه به آبجکت Storage Class فرستاده میشود؛ را میگیرد و از طریق NFS server این حافظه را برای پاد مد نظر فراهم میکند.

به منظور آنکه NFS server را راه اندازی کنیم باید مراحل زیر را انجام دهیم.

الف) sudo apt install nfs-kernel-server برای گره مستر و sudo apt install nfs-common برای گره های کارگر

ب) mkdir /svr/nfs/kubedata && sudo chown nobody: /svr/nfs/kubedata

ج) Sudo systemctl enable nfs-server && sudo systemctl start nfs-server

د) در فایل etc/exports/ خط زیر را وارد میکنیم.

/srv/nfs/kubedata \*(rw,sync,no\_subtree\_check,no\_root\_squash,no\_all\_squash,insecure)

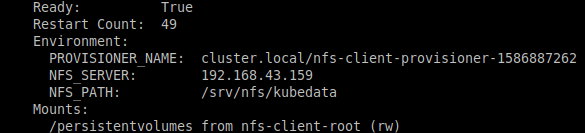
ه) sudo exportfs -rav

حال برای اینکه NFS-server کار میکند باید از گره کارگر دستور زیر وارد کنیم .

Mount -t nfs 192.168.43.159:/svr/nfs/kubedata /mnt

آدرسی که می بینیم آدرس گره مستر ما است که nfs-server در آن نصب شده است.

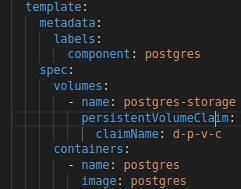
بعد از آنکه این مراحل را انجام دادیم و پاد nfs-provisioning در کلاستر ما اجرا میشود که با دستور kubectl pod describe وضعیت NFS-server و این پاد را مشاهده میکنیم.



شکل ۱۵ - وضعیت پاد nfs-client-provisioner

همان طور که میبینیم این پاد به NFS-server متصل است و مسیر persistentvolumes/ در داخل کانتینر به سرور /svr/nfs/kubedata متصل شده است و درخواست های حافظه ای که از storageclass میگیرد را در این مسیر برای پاد های مختلف ایجاد میکند.

و نکته آخر برای آنکه از این فراهم کننده حافظه استفاده کنیم باید در آبجکت persistent Volume Claim در قسمت StorageClass به جای manual؛ nfs-client قرار بدهیم تا این فراهم کننده بتواند به صورت خودکار برای ما حافظه مورد نیاز را تامین کند. همچینی در آبجکت فایل Deployment باید به آبجکت Persistent volume claim مد نظر خود در قسمت spec اشاره کنیم تا این حافظه برای پاد مد نظر درست شود.



شکل ۱۶- مشخص کردن PVC برای پاد ای که میخواهد حافظه پایدار داشته باشد

در اینجا ما برای سرویس پایگاه داده postgres از persistent volume استفاده کرده ایم تا داده ها از بین نروند. برای آنکه بتوانیم از این حافظه استفاده کنیم باید در آبجکت فایل deployment سرویس postgres اسم درخواست حافظه ای که قبلا درست کردیم را بیاوریم.

3 جمع بندی :

در این فصل ما کلاستر سه گره ای خود را به صورت محلی راه اندازی کردیم و سپس کانفیگ فایل های مربوط به آبجکت های پلتفرم کوبرنتیز را نوشتیم تا این پلتفرم بتواند سرویس های ما را به صورت خودکار مدیریت کند.

**فصل پنجم: روش های خودکار سازی در پلتفرم کوبرنتیز**

در این فصل ما انواع روش های خودکار سازی در کوبرنتیز را معرفی میکنیم و توضیح میدهیم. ولی روشی که ما برای خودکارسازی در این پروژه استفاده میکنیم روش سوم که خودکار سازی سرویس ها است را استفاده میکنیم. روش اول به دلیل آنکه منابع سخت افزاری ما محدود است؛ امکان استفاده از این روش وجود ندارد و روش دوم به دلیل آنکه با روش سوم تداخل دارد و در حال حاضر این امکان وجود ندارد که روش دوم و سوم را با هم استفاده کنیم؛ نمیتوانیم از روش دوم در این پروژه بهره ببریم.

# 1 خودکار سازی تعداد گره ها (cluster auto-scaler):

یکی از ابزار کوبرنتیز(Kubernetes) است که به ما این اجازه را میدهد سایز کلاستر خود را که شامل تعدادی گره است؛ افزایش یا کاهش دهیم . این خودکار ساز در لایه Infrastructure کار میکند و هر موقع پاد های ما دچار کمبود منابع شدند و گره ای وجود نداشت که این پاد تخصیص یابد؛ آنگاه این ابزار یک گره به کلاستر اضافه میکند. و هنگامی که منابع ما از یک حدی به بعد مورد استفاده قرار نگرفتند ؛ گره ها کم میشوند تا در مصرف منابع سخت افزاری صرفه جویی شود. این ابزار کلاستر ما را انعطاف پذیر و مقیاس پذیر میکند. به منظور اینکه این ابزار درست کار کند اقداماتی باید صورت پذیرد.

الف) سایز گره ها باید از لحاظ مقدار محاسبات و حافظه یکسان باشد تا این ابزار بتواند کار خودکار سازی را برای کلاستر انجام دهد.

ب) درخواست های محاسباتی و حافظه ای برای هر پاد باید مشخص باشد به این منظور که این ابزار بتواند مقدار استفاده (Utilization) از این گره را محاسبه کند و در صورت نیاز تعداد را کاهش یا افزایش دهد در غیر این صورت این ابزار دچار مشکل خواهد شد.

ج) باید یک حدی را مشخص کنیم که تعداد گره ها از این مقدار پایین نیایند و باعث نشود که سرویس های مهم و اساسی ما کم یا از بین بروند و معیار دسترس پذیری بالا (High availability) سرویس ها خدشه دار شوند. در کوبرنتیز ما با استفاده از PodDistruptionBugdet می توانیم پاد ها را به گونه ای تنظیم کنیم که از یک مقدار مقدار مشخص کمتر نشوند و همیشه یک تعداد مشخص در حال اجرا باشند و هنگامی که ادمین کلاستر قصد این را داشت که گره ای را خاموش کند که این مقدار پاد کمتر از حد مجاز شود؛ این اجازه را نمیدهد.

د) درخواست های محاسباتی و حافظه ای (CPU and Memory Requests) که برای پاد ها مشخص میکنیم باید نزدیک به واقعیت باشد ( نه خیلی بیشتر و نه خیلی کمتر) تا این ابزار دچار اشتباه

محاسباتی نشود و منابع اضافی بیشتر یا کمتر استفاده نکند. این کار باعث میشود تا منابع به طور بهینه مصرف شود. برای اینکه درخواست های ما نزدیک به واقعیت باشد باید یا از ابزار VPA ( خودکار سازی تخصیص منابع) استفاده کنیم یا خودمان بر اساس عملکرد سرویس در مواقع مختلف و زیر بار های مختلف تشخیص دهیم که چه میزان منابع برای این سرویس لازم است.

با توجه به اینکه ما در این پروژه از سه گره استفاده میکنیم و این سه گره را به صورت Local ( محلی) با استفاده از ابزار Kubeadm راه اندازی کردیم و از فراهم آور های ابری (cloud providers) مانند AWS , Google Cloud و Azure استفاده نکرده ایم؛ امکان استفاده از این ابزار وجود ندارد.

2 خودکار سازی تخصیص منابع (VPA):

این خودکار ساز با توجه به منابعی که یک پاد مصرف میکند؛ درخواست های محاسباتی و حافظه ای مناسبی را میتواند هم پیشنهاد بدهد یا خودش آن درخواست ها را برای پاد تنظیم کند. بنابراین زمانی که مصرف یک پاد زیاد باشد به آن نسبت درخواست ها را افزایش میدهد و بالعکس زمانی که مصرف کم باشد درخواست ها را کم میکند.این باعث میشود که منابع ما بهینه مدیریت شوند.

بخش مهم این خودکارساز Recommender ( پیشنهاد دهنده) است که با استفاده از Metrics-server که در بخش بعدی توضیح داده خواهد شد؛ میزان مصرفی محاسباتی و حافظه ای را میگیرد و بر اساس آن به ما بهترین پیشنهاد خود را میدهد.

- نصب:

برای نصب این خودکار ساز ابتدا باید Metrics-server را نصب داشته باشید که در بخش بعدی نصب آن توضیح داده خواهد شد.

سپس از این آدرس با استفاده از git آن را clone کنید:

git clone <https://github.com/kubernetes/autoscaler.git>

و با رفتن به پوشه vertical-pod-autoscaler دستور را اجرا کنید:

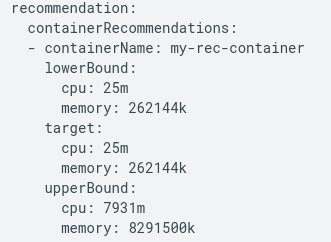
./hack/vpa-up.sh

حال با استفاده از فایل Yaml زیر میتوانیم VerticalPodAutoscaler خود را بسازیم.



شکل۱ - فایل Yaml برای ساختن خودکار ساز

همانطور که در شکل بالا مشاهده می کنیم این خودکار ساز برای یک deployment تنظیم شده است و چون updateMode غیر فعال است؛ این خودکار ساز تنها مقادیری که برای Cpu و Memory مناسب هست را به ما پیشنهاد میدهد و این پیشنهاد را برای سرویس ما اعمال نمیکند.



شکل۲ - خروجی خودکارساز تخصیص منابع (VPA)

این خودکار ساز همانطور که در شکل بالا آمده است؛ مقادیر حد پایین ؛ حد مطلوب و حد حداکثر را برای ما پیشنهاد داده است. حال اگر updateMode فعال شود مقدار مطلوب را در قسمت درخواست های محاسباتی و حافظه ای اعمال میکند.



شکل۳ - اعمال مقیاس پذیری تخصیص منابع برای پاد مد نظر

و در اینجا بعد از آنکه این خودکار ساز پاد را دوباره میسازد؛ این درخواست ها را در پاد مد نظر ما اعمال میکند و دیگر لازم نیست که ما بخواهیم خودمان به طور دستی این درخواست ها را برای پاد ها مشخص کنیم.

مشکلی که در این خودکار ساز وجود دارد این است که چون هنوز در حالت آزمایشی قرار دارد با خودکارساز سرویس ها هنوز سازگار نیست و نمی توان از هر دو همزمان هنگامی که از معیار های محاسباتی و حافظه ای برای مقیاس پذیری استفاده میکنیم؛ استفاده کرد و فقط این دو زمانی با هم کار میکنند که ما از custom metrics استفاده کنیم که معیار هایی هستند که خودمان آنها را تعریف میکنیم و با استفاده از آنها سرویس های خود را مقیاس پذیر میکنیم.

چون کلاستر ما کوچک است و خودمان با استفاده از میزان مصرفی پاد ها در زمان های مختلف میتوانیم درخواست های محاسباتی و حافظه ای مناسب را برای پاد های خود مشخص کنیم. به علاوه به دلیل آنکه میخواهیم از معیارهای محاسباتی و حافظه ای استفاده کنیم؛ در نتیجه از این خودکار ساز در پروژه استفاده نخواهیم کرد.

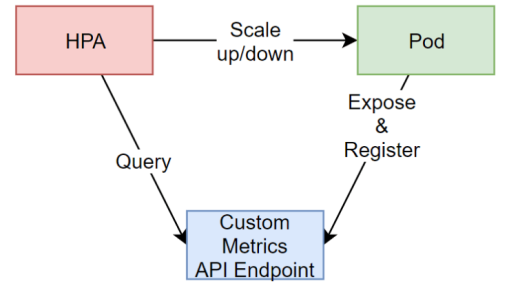
## 3 خودکار سازی سرویس ها (HPA):

3-1 Metrics server

بخش سوم و بخش مهم این پروژه در این قسمت است که ما میخواهیم سرویس های خود را بر اساس معیار های مختلف مقیاس پذیر کنیم به منظور آنکه سرویس های خود را Responsive (پاسخگو) نگه داریم و از تاخیر زیاد و بار زیاد بر روی یک سرویس جلوگیری شود.

برای اینکه ما بتوانیم پادهای خود را بر اساس معیار های محاسباتی و حافظه ای (Cpu and Memory metrics) مقیاس پذیر کنیم؛ باید از یکی از ابزار هایی که میتوان در کنار kubernetes-api نصب کرد Metrics-server است که اطلاعات را که از گره ها با استفاده از Kubelet که در هر گره وجود دارد

جمع آوری میکند و از طریق (API (metrics.k8s.io در دسترس HPA قرار میدهد که بتواند میزان مصرفی پاد ها رصد کند و هر موقع میزان مصرفی محاسباتی یا حافظه ای ما از حد مشخص شده رد شد؛ این خودکار ساز شروع به افزایش پاد ها کند.



شکل۴ - نحوه عملکرد خودکار ساز

این شکل یک شمای کلی از کارکرد HPA بیان میکند که در این خودکار ساز با استفاده از metrics server که معیار ها را از هر گره استخراج میکند؛ کار مقیاس پذیری را انجام میدهد.

HPA برای آنکه مشخص کند چه تعداد پاد لازم است تا افزایش یا کاهش دهد از الگوریتم ساده زیر استفاده میکند.

desiredReplicas = ceil[currentReplicas \* ( currentMetricValue / desiredMetricValue )]

که تعداد پاد های فعلی را در تقسیم مقدار الان بر مقدار مطلوب ضرب میکند. برای آنکه بیشتر مشخص شود. به طور مثال اگر مقدار مطلوب ما برای یک پاد ۱۰ درخواست در ثانیه باشد و مقدار حال حاضر ۲۰ درخواست بر ثانیه است باید پاد ها ار دو برابر کند و اگر از این مقدار کمتر شد پاد ها را کاهش میدهد.

در این خودکار ساز یک مفهومی به Cooldown Period وجود دارد به این معنی که یک زمانی طول میکشد که این خودکار ساز پاد ها را افزایش دهد یا کاهش دهد. این به این منظور است که شاید پیک های گذرا رخ دهد و لازم نباشد که مقیاس پذیری صورت بگیرد. همین طور در موقع کاهش ممکن است بار زیادی در حال حاضر وجود دارد ولی یک وقفه در بار بیفتد و دوباره بار زیاد ادامه پیدا کند. پس این زمان مهم است که این خودکار ساز از کم شدن بار یا زیاد شدن بار اطمینان حاصل کند و بیهوده کار مقیاس پذیری را انجام ندهد. مقدار پیش فرض ۵ دقیقه است.

حال به سراغ راه اندازی این خود کار ساز با استفاده از metrics server میرویم.

3-1-1 نصب Metrics server

برای راه اندازی metrics-server از پکیج منیجر Helm استفاده میکنیم که این پکیج منجیر Third-party software است که به کلاتسر ما دسترسی پیدا میکند و سرویس هایی که نیاز داریم را برای ما نصب میکند و کار را برای ما بسیار آسان میکند و لازم نیست کلی کانفیگ فایل و آبجکت های مختلف را در کلاستر خود نصب کنیم.

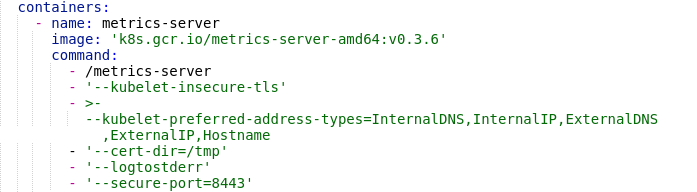
ابتدا برای نصب این پکیج باید یک سری مقادیر را تغییر دهیم به این منظور که در حین نصب دچار مشکل نشویم. دستور زیر را وارد میکنیم و مقادیر را تغییر میدهیم.

helm show values stable/metrics-server > /tmp/metrics-server.values

دو مقدار زیر باید به این گونه باشند.

الف) hostNetwork: true

ب)



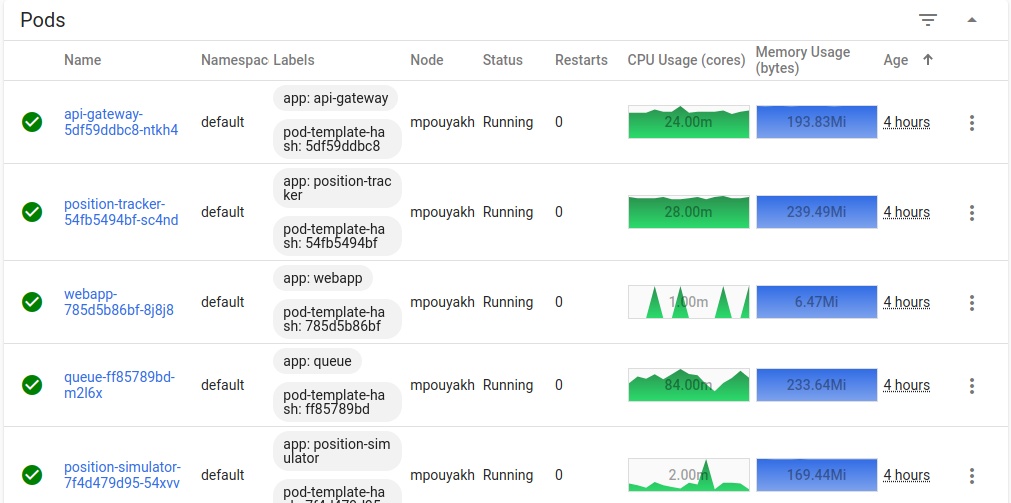
شکل۵ - اضافه کردن پارامترهای اضافی به فایل Yaml سرویس Metrics server

در قسمت command باید خط دوم و سوم را اضافه کنیم.

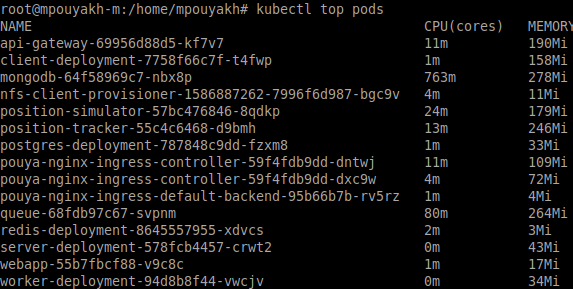
بعد از این تغییرات نوبت به نصب این پکیج میرسد که با استفاده از دستور زیر metrics server را نصب میکنیم.

helm install metrics-server stable/metrics-server --namespace metrics --values /tmp/metrics-server.values

حال میتوانیم با استفاده از داشبورد کوبرنتیز مقدار مصرفی محاسباتی و حافظه ای برای هر پاد را مشاهده کنیم. و در بخش command line با استفاده از دستور kubectl top pods میزان مصرفی را مشاهده کنیم که در صفحه بعدی شکل های داشبورد و command line آمده است.



شکل۶ - نمای کلی از نمایش معیار های محاسباتی و حافظه ای در داشبورد کوبرنتیز



شکل۷ - نمایش معیار های محاسباتی و حافظه ای در ترمینال

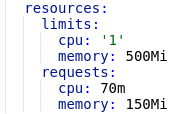
همانطور که در شکل های بالا مشاهده می کنیم میزان مصرفی هر پاد مشخص است که برای قسمت cpu منظور از m ؛ milicore است که هر یک cpu برابر است باm 1000.

حال با انجام این بخش؛ سراغ مرحله بعد که کار با HPA است میرویم و برای یک سرویس این مقیاس پذیر کردن را انجام میدهیم.

برای اینکه خودکار ساز ما در این بخش کار کند؛ باید برای پاد هایی که میخواهیم مقیاس پذیری را انجام دهیم درخواست های محاسباتی و حافظه ای مناسبی را برای پاد های خود تعیین کنیم و چون در این بخش که مقیاس پذیر کردن بر اساس میزان مصرفی cpu و memory است ؛ نمی توانیم از VPA استفاده کنیم باید خودمان بر اساس منابعی که در کل در اختیار داریم و سابقه مصرفی پاد مورد نظر؛ یک درخواست نزدیک به واقعیت تعیین کنیم.

برای تست این مقیاس پذیری ما از client-deployment استفاده میکنیم که بخش فرانت اند وب اپلیکیشن ما است.

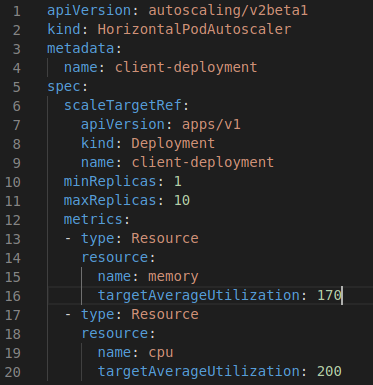
حال با بررسی مقدار مصرفی این پاد و منابعی که در اختیار داریم ما درخواست ها و محدودیت های زیر را برای این پاد در نظر گرفته ایم.



شکل۸ - تعیین درخواست های محاسباتی و حافظه ای برای سرویس مد نظر

3-1-2 ساختن خودکار ساز برای metrics server

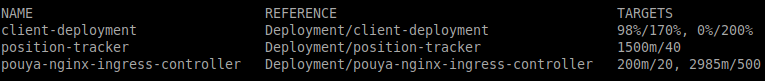
سپس فایل Yaml این خودکار ساز را نوشتیم که به صورت زیر است:



شکل۹ - فایل Yaml برای ساختن خودکار ساز

همانطور که مشخص است در خط اول ابتدا API مد نظر را مشخص کردیم که با استفاده از این API به این خودکار ساز می توانیم دسترسی پیدا کنیم و در قسمت بعدی مشخص میکنیم که چه نوع آبجکتی (Object) میخواهیم درست کنیم. بخش مهم دیگر این فایل قسمتی است که معیار ها را مشخص میکنیم. در این فایل دو معیار cpu و memory را تعیین کردیم که مقدار مطلوب ما برای cpu مساوی دو برابر (%200) درخواستی که در صفحه قبل مشخص کردیم که برابر 140m میشود. و برای memory ما ۱۷۰ درصد مقدار درخواستی که تعیین کردیم؛ برابر 255MB است.

بعد از ساختن این آبجکت در کوبرنتیز؛ خودکار ساز ما به این شکل در خواهد آمد.



شکل۱۰ - نتیجه خودکارساز برای Metrics server

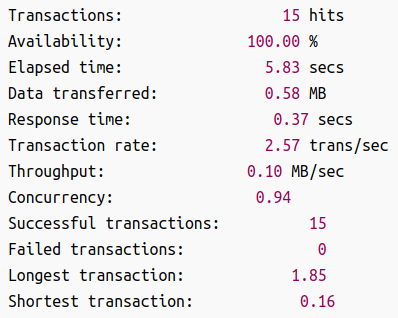
در خط اول این شکل مشاهده می کنیم که در قسمت Targets دو معیار ما آورده شده اند و تعداد Replica برابر ۱ است چون مقدار فعلی کمتر از مقدار تعیین شده است. بعد از آنکه بار بر روی این پاد اضافه کردیم مشاهده می کنیم به صورت خودکار؛ کوبرنتیز تعداد پاد ها را افزایش خواهد داد و سعی میکند مقدار فعلی را به زیر مقدار تعیین شده ببرد. همچنین بعد از آنکه بار کم شد این خودکار ساز بعد از ۵ دقیقه که زمانی است که خودکار ساز صبر میکند تا تعداد پاد ها را کاهش دهد؛ تعداد پاد ها را کم میکند.

3-2 تولید بار (Load Generator):

حال سراغ تولید بار می رویم که برای این کار ما از ابزار siege استفاده میکنیم. برای اینکه بتوانیم تشخیص دهیم که توانایی وب اپلیکیشن یا سرویس ما چقدر است و چه میزان توانایی پاسخگویی دارد باید بر روی آن بار تولید کنیم. با استفاده از این ابزار درخواست های HTTP را به میزانی که مد نظر است در یک بازه زمانی مشخص شده برای پاد ما میفرستد. بعد از نصب این پکیج در سیستم خود آدرس پاد خود را به این تولید کننده بار (load generator) میدهیم و به طور مثال۲۰ کاربر همزمان درخواست های خود را برای این سرویس به مدت ۱۰ دقیقه میفرستیم و بعد نتیجه را مشاهده کنیم.

siege -q -c 20 -t 10 <http://10.102.108.164:3000>

آدرس پاد ما در دستور بالا آمده است که بر روی پورت ۳۰۰۰ میدهد. و نتایج این تولید کننده بار برای مثال به صورت زیر است.



شکل۱۱ - نتیجه تولید کننده بار siege

در قسمت اول نشان میدهد که چه میزان در کل درخواست انجام شده است. در قسمت دوم میزان دسترس پذیر بودن سرویس ما را نشان میدهد که چه میزان از این درخواست ها جواب ۲۰۰ گرفته شده است. بخش مهمی که باید در این بخش در رابطه با این تولید کننده بار ذکر کرد؛ میانگین زمان پاسخ کل درخواست ها است (Response time) که در قسمت پنجم شکل آمده است و بخش Concurrency تعداد کاربری که همزمان به سرویس ما متصل هستند را نشان میدهد. بخش آخر به طولانی ترین و کوتاه ترین درخواست های ما از کل درخواست هایی که فرستاده شده است که در دو قسمت آخر آمده است؛ اشاره میکند.

همچنین این تولید کننده بار این امکان را به ما داده است که بتوانیم با استفاده از کانفیگ فایل این تولید کننده بار که در این مسیر است) siege/siege.conf. (~/؛ قابلیت های مختلف این ابزار را استفاده کنیم. به طور مثال چند نمونه از قابلیت های مهم این ابزار را در قسمت زیر اشاره میکنیم.

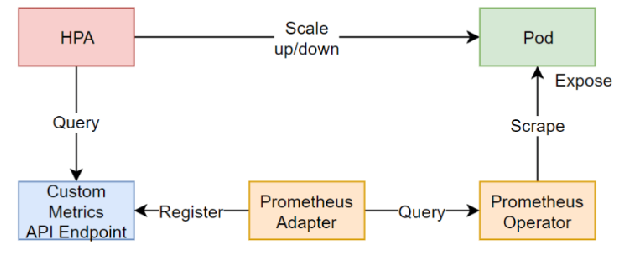
اگر ما در این کانفیگ فایل بخش parser را فعال کنیم یعنی مساوی true قرار دهیم؛ این تولید کننده بار فقط یک درخواست HTTP به همان مسیری که مشخص کردیم میدهد و دیگر بقیه بخش های این وب پیج را که شامل فایل های Javascript و CSS و عکس ها است؛ درخواست نمیدهد. همچنین می توانیم محدودیت تعداد کاربر همزمان را که به طور پیش فرض ۲۵۵ است را تغییر دهیم و افزایش دهیم. در بخش دیگر این کانفیگ فایل میتوانیم با استفاده از فعال کردن قسمت verbose ؛ نتیجه درخواست ها را در command line خود مشاهده کنیم. این تولید کننده بار بین درخواست ها به طور پیش فرض بین یک تا سه ثانیه به طور رندم فاصله ایجاد میکند که در بخش delay میتوانیم این زمان را افزایش دهیم. ولی برای آنکه ما عملکرد سرویس خود را بهتر تست کنیم و توان سرویس خود را ارزیابی کنیم این تاخیر را صفر در نظر گرفتیم.

3-3 Custom metrics

در این بخش ما قصد این را داریم که اپلیکیشن خود را بر اساس معیار های بیشتری بتوانیم مقیاس پذیر کنیم. در بخش قبل ما با استفاده از metrics server تنها میتوانستیم معیار های محاسباتی و حافظه ای پاد ها را استخراج کنیم و از طریق Resource metrics API در دسترس HPA قرار دهیم. ولی اگر بخواهیم سرویس های خود را بر اساس معیار های دیگری مقیاس پذیر کنیم باید سراغ راه حل های دیگر برویم.

راه حلی که برای این منظور وجود دارد ؛ Prometheus adapter است که یک نوع Metric API server است که وظیفه آن را دارد که معیار هایی که Promethues به عنوان معیار جمع آور (Metrics collector) انجام میدهد را از طریق Custom Metrics API در اختیار HPA قرار بدهد تا بتواند بر اساس معیار ها کار مقیاس پذیری را انجام دهد. همچنین Prometheus adapter یک فایل کانفیگ به عنوان ورودی دریافت میکند که در آن فایل مشخص شده است ما چه معیار هایی را میخواهیم و چگونه آن را به گونه ای که مورد مطلوب ما هست تحویل ما بدهد.

مورد بعدی که وجود دارد خود Prometheus معیار هایی از نود ها و پاد ها جمع آوری میکند ولی بعضی از معیار ها مخصوص خود اولیکیشن هستند و باید در اپلیکیشن تنظیم شوند و آن معیار ها را در اختیار Prometheus قرار دهند.

شکل ۱۲ - نحوه ارتباطات اجزا مختلف برای استخراج Custom metrics

در این شکل به خوبی نشان داده شده است که چگونه ما با استفاده از Prometheus adapter میخواهیم کار مقیاس پذیری را انجام دهیم و بخش های مختلف چگونه با هم ارتباط برقرار میکنند. همانطور که مشاهده می شود؛ Prometheus معیار ها را از پاد ها استخراج میکند و به صورت زمانی میتواند به ما نشان دهد. سپس Prometheus adapter این معیار ها را به گونه ای که ما میخواهیم در میاورد و از طریق API مشخص شده در شکل برای HPA در دسترس قرار میدهد.

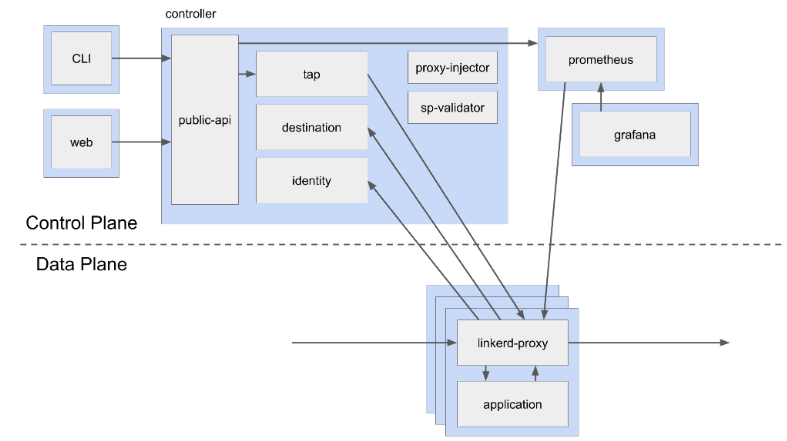
حال به سراغ مرحله بعد می رویم که در صفحه قبل اشاره کردیم. برای اینکه بعضی از معیار ها را بتوانیم تعریف کنیم و بر اساس آنها مقیاس پذیری را انجام دهیم باید کد اپلیکیشن خود را به گونه ای تغییر دهیم که بتواند این معیار ها را در اختیار Prometheus بگذارد. ولی ما با استفاده از Linkerd که یک Service Mesh برای کوبرنتیز است؛ میخواهیم بدون آنکه تغییری در سرویس های بدهیم بعضی از معیار های مفید را برای مقیاس پذیری استفاده کنیم.

3-3-1 نصب و ساختار linkerd

ابتدا توضیح مختصری از اینکه این Service Mesh چگونه کار میکند داده میشود سپس سراغ استفاده از آن میرویم.

ساختار این سیستم به این گونه است که دو بخش دارد. بخش اول Control Plane نام دارد که مسئول مدیریت پروکسی هایی است که در کنار هر سرویس قرار میگیرد است و مسئول ارتباطات این پروکسی ها است. مسئولیت های دیگر این بخش مدیریت؛ جمع آوری اطلاعات از پروکسی ها؛ فراهم کردن ارتباطات بر اساس تکنولوژی TLS و فراهم کردن API هایی برای ادمین کلاستر است که بتواند داده هایی که از این پروکسی ها جمع آوری شده دسترسی پیدا کند و بتواند تغییراتی را در این پروکسی ها اعمال کند.

بخش دیگر این سیستم؛ data plane است که مربوط به پروکسی هایی است که در کنار سرویس های ما قرار میگیرد و درخواست هایی که به سرویس ها میشود را دریافت میکند و به سرویس میفرستد سپس بر اساس جواب هایی که میگیرد تصمیماتی میتواند بگیرد و آن ها را انجام دهد به طور مثال اگر جواب نگرفت دوباره درخواست را بفرستد و نتیجه جواب ها را ذخیره کند یا زمان تاخیر هر سرویس را محاسبه کند و کار های دیگر. در شکل زیر شمای کلی این سیستم را مشاهده میکنیم.



شکل۱۳ - ساختار کلی سیستم Linkerd

همانطور که در شکل آمده است Control Plane بخش های مختلفی دارد که یکی از بخش های مهم آن که ما با آن کار داریم بخش Promethues است که اطلاعلات را از پروکسی ها دریافت میکند و در اختیار ما میگذارد. بخش دیگری که ما استفاده میکنیم سیستمی به نام grafana است که ابزار قوی ای برای تصویر سازی داده ها است و با استفاده از این ابزار میتوانیم داده های جمع آوری شده از طریق Prometheus را به طرز مفید و زیبایی ببینیم. پس linkerd این دو ابزار مهم را هم برای ما نصب میکند و لازم نیست تا جداگانه این دو ابزار را نصب و راه اندازی کنیم.

حال به سراغ نصب linkerd میرویم:

ابتدا برای آنکه این سیستم را نصب کنیم؛ باید command line مخصوص linkerd را نصب کنیم تا بتوانیم به linkerd دسترسی پیدا کنیم و عملیات های مختلف انجام دهیم.

با دستور زیر میتوانیم این CLI را نصب کنیم.

curl -sL https://run.linkerd.io/install | sh

سپس باید به path با استفاده از دستور زیر اضافه کنیم.

export PATH=$PATH:$HOME/.linkerd2/bin

بعد از انکه Linkerd به کلاستر ما دسترسی پیدا کرد و با استفاده از دستور زیر همه ی پیش نیاز های کلاستر ما را بررسی کرد و همه پیشنیاز ها آماده بود سراغ مرحله آخر که اضافه کردن پروکسی linkerd به سرویس ها است.

Linkerd check --pre

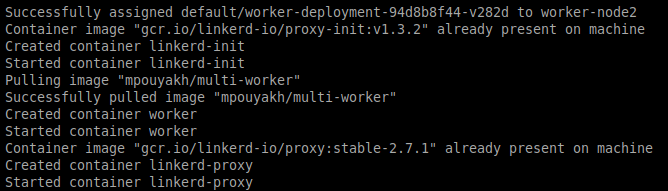
با استفاده از دستور زیر linkerd را نصب میکنیم:

linkerd install | kubectl apply -f -

بعد از نصب linkerd میتوانیم با دستور زیر این پروکسی را در سرویس خود اضافه کنیم.

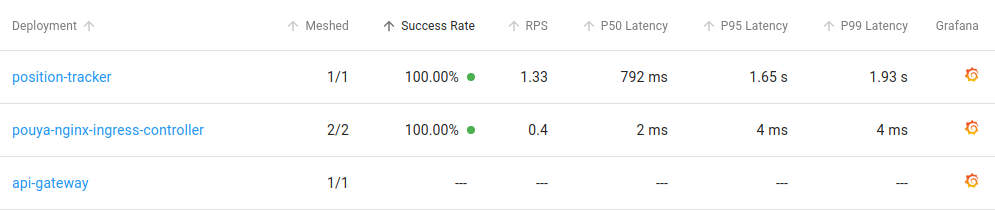
kubectl get -n default deploy/worker-deployment -o yaml | linkerd inject - | kubectl apply -f -

و در شکل زیر اضافه شدن این پروکسی را در سرویس خود میتوانیم مشاهده کنیم.

شکل۱۴ - اضافه شدن پروکسی linkerd به پاد مد نظر

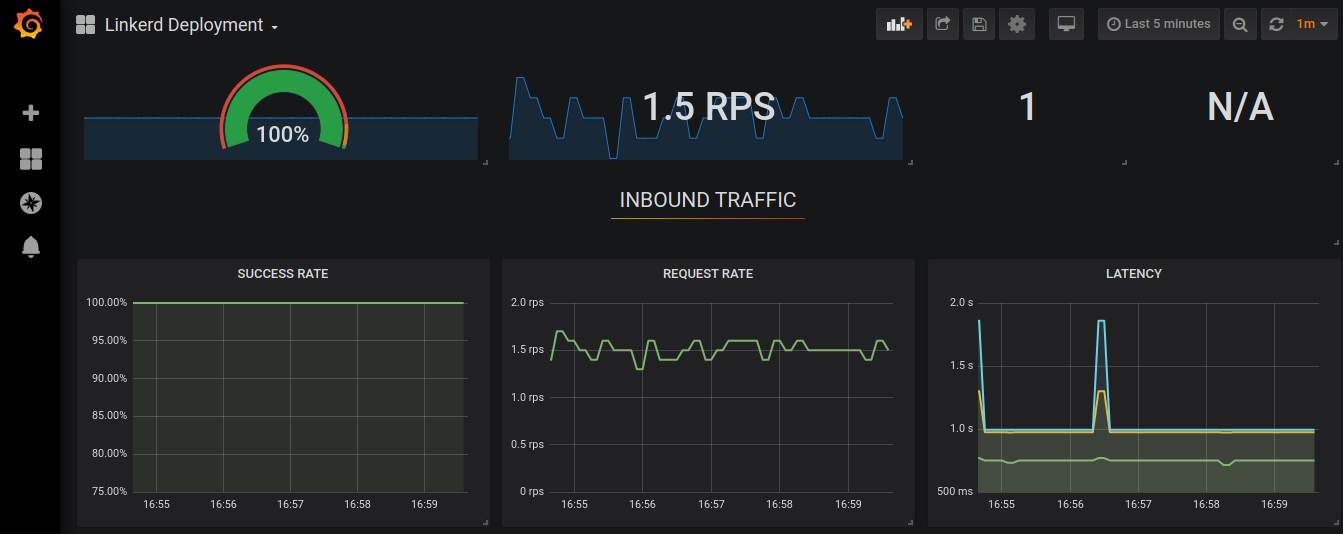
Linkerd همچنین یک web UI دارد که از طریق آن میتوانیم سرویس های اضافه شده را مشاهده کنیم. در شکل زیر این قابلیت linkerd را هم مشاهده کنیم.

با دستور "linkerd dashboard -- port 30000 "میتوانیم این دشبورد را بالا بیاوریم.



شکل۱۵ - نمای داشبورد Linkerd

با استفاده از linkerd میتوان میزان تاخیر هر سرویس به علاوه اینکه چه میزان درخواست در یک ثانیه دریافت میکند را مشاهده کنیم و از طریق grafana میتوان تاریخچه این میزان درخواست ها و تاخیر ها را مشاهده کرد که در شکل زیر خواهیم دید.

شکل۱۶ - نمای دشبورد Grafana

سه نوع تاخیری که وجود دارد به این منظور است که مشخص کند که این تاخیر برای چند درصد از مواقع درست است و وقتی تاخیر P99 است؛ این به این منظور است که تنها یک درصد درخواست ها تاخیرشان پایین این تاخیر نوشته شده است و P95 هم به همین شکل یعنی فقط ۵ درصد درخواست ها تاخیرشان از این میزان کمتر از این تاخیر ذکر شده است.

3-3-2 معیار های تاخیر در پاسخ (Response latency) و تعداد درخواست در یک ثانیه (RPS):

حال میخواهیم بر اساس یک سری معیار های مفید تر دیگری کار مقیاس پذیری را انجام دهیم. درست است که معیار محاسباتی و حافظه ای تا حدودی اطلاعات خوبی را به ما درباره اینکه چقدر بار بر روی یک سرویس ما گذاشته میشود؛ ولی خیلی دقیق نیست و وقتی میزان مصرفی cpu بالا میرود شاید دلایل دیگری برای این بالا رفتن بار وجود دارد و تنها بار اضافه شده بر روی سرویس ما نیست. ممکن است این بالا رفتن cpu به دلیل این است که بعضی از باورهای ما Memory bound و IO bound هستند. همچنین cpu ها مختلف هستند و هر کدام ممکن است مقدار مصرفی که نشان دهند متفاوت باشد و این معیار نسبی است و خیلی دقیق نیست. حال معیاری که میتوانیم رویش حساب کنیم و مقیاس پذیری را انجام دهیم معیار تاخیر در جواب (response latency) یک سرویس است که به ما نشان می دهد چه مدت طول می کشد تا سرویس ما به کاربر جواب بدهد. شرکت ها و کمپانی های بزرگ همه در صدد کم کردن میزان این تاخیر هستند تا بتوانند به رضایت مشتری را جذب کنند. پس این معیار بسیار اساسی و مهم است. شما اگر یک سرویس با شکل و شمایل بسیار زیبا هم داشته باشید ولی زمان پاسخ زیاد باشد؛ مشتری راضی نخواهد بود و امکان از دست دادن آن مشتری زیاد است.

معیار دیگری که میتواند مفید باشد تعداد درخواست هایی که در یک ثانیه (RPS)به سرویس ما وارد میشود چقدر است و بر اساس چه میزان بار ما آن تاخیر را دریافت میکنیم. همچنین بر اساس این معیار هم میتوانیم کار مقیاس پذیری را هم انجام دهیم.

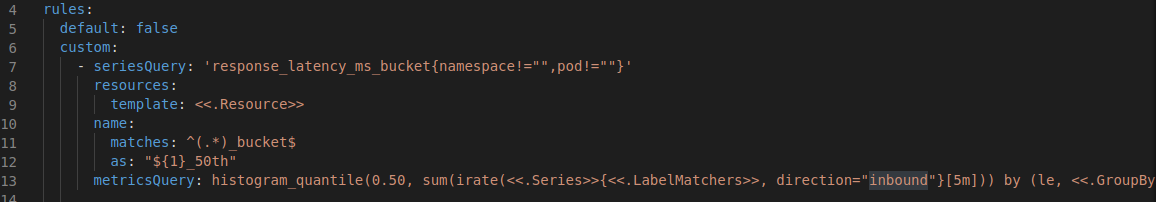
حال بعد از آنکه این دو معیار معرفی شد؛ سراغ استخراج این دو معیار با استفاده از Prometheus adapter میرویم که همان طور در بخش های قبل گفته شد مسئول گرفتن اطلاعات از Prometheus است و مرتب کردن اطلاعات به گونه که ما در کانفیگ فایل مشخص کردیم سپس با در دسترس قرار دادن اطلاعات از طریق API مربوطه برای HPA؛ کار مقیاس پذیری را انجام میدهیم.

کانفیگ فایل به صورت زیر است :



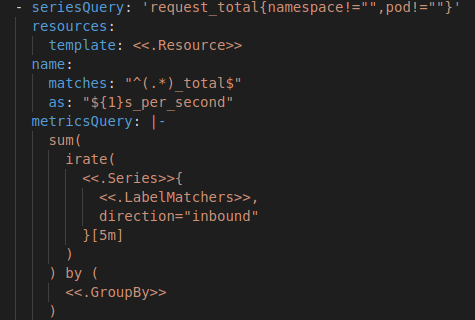
شکل۱۷ - معرفی سرویس Prometheus به Prometheus adapter

در بخش اول ما سرویس prometheus خود را معرفی میکنیم تا این adapter بتواند اطلاعات خود را از آن بگیرد.

شکل۱۸ - بخش مشخص کردن معیار های مد نظر در کانفیگ فایل Promethues adapter

در این بخش ما قوانین ما rule های خود را مینویسیم و معیاری که میخواهیم را مشخص میکنیم. همچنین این اطلاعات به چه صورت به ما داده شود را مشخص میکنیم. با استفاده از بخش seriesQuesry مشخص میکنیم که کدام معیار را از Prometheus میخواهیم استخراج کنیم و میتوانیم اسم این معیار را با استفاده از بخش as تغییر میدهیم. در بخش metricsQuery ما با استفاده از تابع historgram\_quantile میتوانیم تاخیر p50 را همانطور که در بخش قبل اشاره کردیم؛ میتوانیم محاسبه کنیم. با استفاده از تابع sum هم این داده ها را در بازه زمانی ۵ دقیقه جمع میکنیم و به تابع histogram\_quantile به عنوان ورودی میدهیم که تاخیر را محاسبه کند. برای تاخیر های دیگر هم به همین شکل عمل میکنیم و فقط باید اسم معیار را تغییر بدهیم.

برای معیار تعداد درخواست در یک ثانیه هم به همین گونه ای که در شکل نشان داده شد عمل میکنیم و فقط تابع histogram\_quantile را ندارد.



شکل۱۹ - تعیین معیار درخواست در ثانیه و مشخص کردن نحوه دریافت اطلاعات این معیار

بعد از آنکه این کانفیگ فایل را نوشتیم حال سراغ نصب Prometheus adapter با استفاده از پکیج منیجر Helm میرویم که دستورش به شکل زیر است.

helm --namespace linkerd install stable/prometheus-adapter -f hpa/prometheus-adapter.yml

این adapter را در همان جایی که بخش های مختلف linkerd نصب شده است؛ نصب میکنیم و فایل کانفیگ را هم در هنگام نصب به این adapter میدهیم.

حال نگاهی بیندازیم به مقادیری که در custom metrics api که HPA از این طریق میتواند به معیار ها دسترسی پیدا کند؛ وجود دارد. با استفاد از دستور زیر میتوانیم این مقادیر را نگاه کنیم.

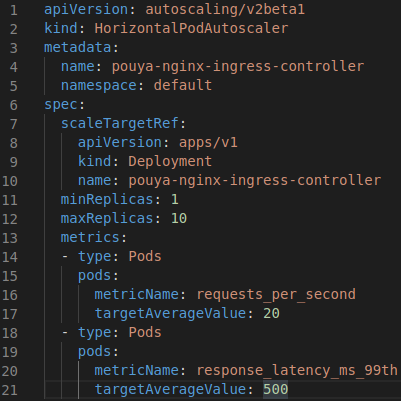
kubectl get --raw /apis/custom.metrics.k8s.io/v1beta1

همان طور که در شکل زیر معلوم است در API میتوانیم اسم این معیار ها را ببینیم و همچنین مقادیرشان را مشاهده کنیم. که در شکل زیر بعضی از این معیار ها آمده است.



شکل۲۰ - مقادیر Custom metrics api

حال بعد از اینکه توانستیم این معیارها را دسترس پذیر کنیم؛ سراغ نوشتن فایل Yaml برای خودکار ساز خود میکنیم.



شکل۲۱ - فایل Yaml برای ساختن خودکارساز

برای این که ما بتوانیم از custom metrics استفاده کنیم؛ از API شده در شکل بالا استفاده میکنیم و این API هم به custom metrics API متصل میشود و اطلاعات معیار ها را دریافت میکند. سپس سرویسی را که میخواهیم مقیاس پذیر کنیم را مشخص میکنیم و API ای که از طریق آن در دسترس هست را می نویسیم که همان apps/v1 است. در مرحله بعد رینج تعدادی که این سرویس ما میتواند مقیاس پذیر شود را مشخص میکنیم که از تعداد یک تا ۱۰ است. با توجه به اینکه در منابع محاسباتی و حافظه ای محدودیت داریم؛ امکان آنکه تعداد بسیار بالا بگذاریم وجود ندارد و تا ۳۰ پاد سیستم ما به خوبی کار میکند و بیشتر منابع حافظه ای و محاسباتی ما اجازه تولید پاد های جدید را نخواهند داد. در مرحله آخر معیار هایی که میخواهیم بر اساس آنها مقیاس پذیری را انجام دهیم را معین میکنیم و سپس حدی را که اگر از آن گذشت کار مقیاس پذیری را انجام دهد را تعیین میکنیم.

این تاخیر بر حسب رویکرد تیم فنی یک سازمان باید تعیین شود و مشخص کنند که چه میزان تاخیر برای سرویسشان مناسب است و از لحاظ رقابتی جه میزان تاخیر مناسب است. در اینجا ما برای تست تا ۵۰۰ میلی ثانیه را زمان مناسبی برای تاخیر در پاسخ سرویس خود در نظر گرفته ایم و اگر از این حد گذشت؛ خودکار ساز ما شروع به افزایش تعداد پاد میکند تا این تاخیر کمتر شود. همچنین معیار دیگری که تعداد درخواست در ثانیه است را هم در نظر گرفته ایم که اگر از این تعداد درخواست در یک ثانیه برای یک پاد بیشتر شد این بار با پاد های اضافه شده تقسیم کند تا عملکرد بهتری از نظر پاسخ دهی داشته باشد.

3-3-3 نحوه انجام تست برای custom metrics

برای تست و بررسی این خودکار ساز بر اساس این معیار ها؛ میخواهیم زیاد شدن بار و افزایش تاخیر و سایر معیار هایی را که از تولید کننده بار دریافت می کنیم را مورد تحلیل و بررسی قرار دهیم؛ که آیا با افزایش تعداد پاد ها توسط خودکار ساز؛ کمکی به کم شدن این تاخیر خواهد کرد و همچنین تعداد بیشتری درخواست را این سرویس ما می تواند پاسخ بدهد. برای این تست ما چند سناریو مشخص را بر اساس محدودیت های نرم افزاری و سخت افزاری که داریم؛ مشخص کردیم. با استفاده از تولید کننده بار خود؛ برای تعداد کاربر ۳۰۰ ؛ ۴۰۰؛ ۵۰۰ و ۷۰۰ میخواهیم رفتار سیستم خود را تحلیل کنیم. و به علاوه در هر مرحله تعداد ماکزیمم پاد ها را برای خودکار ساز محدود میکنیم تا بهتر بتوانیم نسبت افزایش پاد و کم شدن تاخیر و سایر معیار های بدست آمده را بررسی کنیم. تعداد این پاد ها ۱؛ ۳؛ ۶ و ۱۰ در نظر گرفتیم که خودکار ساز ما بر اساس این تعداد پاد ها محدود خواهد شد. به این معنی که برای هر یک از آن چهار تعداد مشخص کاربر ما؛ خودکار ساز خود را به۱ پاد؛ ۳ پاد ؛ ۶ پاد و ۱۰ پاد محدود میکنیم و تا سقف همین تعداد امکان تولید پاد ها را دارد و نه بیشتر. در آخر؛ نتایج هر بخش را جداگانه دریافت میکنیم و بر اساس نتایجی که از تولید کننده بار برای این ۱۶ باری که این تولیدکننده بار را اجرا کردیم گرفتیم؛ میتوانیم رفتار سیستم خود را تحلیل کنیم که این تحلیل و بررسی نتایج در فصل بعدی آمده است.

بر اساس سیستم Linkerd میزان تاخیر پاسخ سرویس خود و تعداد درخواست در ثانیه را میتوانیم دریافت کنیم و بر اساس سیستم Grafana؛ گراف های این معیارها را میتوان برای زمان های مختلف مشاهده کرد. سپس ما برای هر اجرای تولید کننده بار بر اساس تعداد کاربر های متفاوت و محدودیت هایی که برای پاد ها گذاشته ایم؛ دو گراف از تاخیر پاسخ و تعداد درخواست بر ثانیه ذخیره میکنیم. به علاوه تولید کننده بار هم نتایج مختلفی را به ما میدهد که در بخش Load generator این نتایج را معرفی کردیم. بنابراین بر اساس نتایج تولید کننده بار و این دو گراف بدست آمده از Grafana؛ تحلیل های خود را انجام میدهیم.

4 جمع بندی:

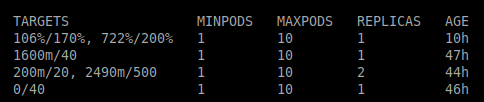
در این فصل ما سه روش اصلی برای خودکارسازی در کوبرنتیز را توضیح دادیم و پیاده سازی روش سوم را که بخش اصلی این پروژه هست را به طور دقیق و مفصل شرح دادیم که چگونه با استفاده از معیار های مختلف سرویس های خود را در پلتفرم کوبرنتیز مقایس پذیر کنیم.

فصل ششم: نتایج خودکار سازی سرویس ها

در این فصل ما نتایج بدست آمده از مقیاس پذیری بر اساس معیارهای مختلف را به نمایش خواهیم گذاشت. که در ابتدا نتایج برای معیار های محاسباتی و حافظ ای را نشان می دهیم. سپس نتایج مقیاس پذیری سرویس ها بر اساس معیارهای زمان پاسخ و تعداد درخواست در ثانیه را به صورت گراف ها و جداول مختلف نشان خواهیم داد و تحلیل و بررسی خواهیم کرد.

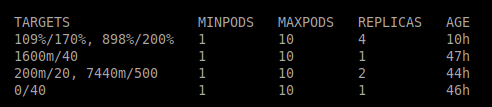
1 نتایج خودکار ساز برای معیار های محاسباتی و حافظه ای

حال نتایج عملکرد این خودکار ساز برای معیار های محاسباتی و حافظه ای را در عکس های زیر مشاهده میکنیم. برای اینکه میزان مصرفی محاسباتی سرویس ما زیاد شود از ابراز siege که قبلا توضیح داده بودیم استفاده میکنیم.



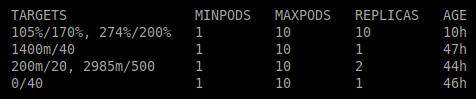
شکل۲۲ - خروجی خودکار ساز

در اینجا مشاهده میکنیم که بار بر روی پاد ما زیاد شده است و تقریبا ۳.۵ برابر حدی که برای خودکار ساز خود تعیین کرده ایم شده است. ولی معیار ما از لحاظ حافظه ای کمتر از حد مجاز است و این مقیاس پذیری بر اساس معیار محاسباتی انجام میپذیرد.



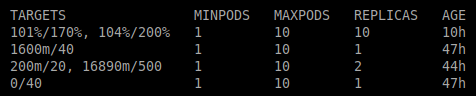
شکل۲۳ - مقیاس پذیری سرویس مد نظر

در مرحله بعدی می بینیم که بار زیاد تر شده است و خودکار ساز تعداد بیشتری از این پاد را میسازد و هنوز مقدار مصرفی پاد ها بیشتر از حد تعیین شده است که یک پاد مجاز است مصرف کند. پس انتظار میرود که خودکار ساز پاد های بیشتری تولید کند.



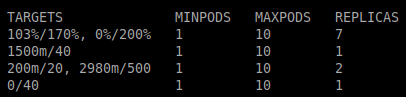
شکل ۲۴ - مقیاس پذیری سرویس و کم شدن بار بر روی هر سرویس

وقتی ماکزیمم تعداد مجاز پاد ها تولید شد؛ میبینیم که درصد مصرفی هر پاد کمتر پایین تر آمده است و بار بر روی این پاد ها بخش شده است و یک پاد بار زیادی را تحمل نمیکند. و در شکل زیر خواهیم دید که میزان مصرفی هر پاد از مقدار تعیین شده کمتر شده است.



شکل۲۵ - تقسیم بار بر روی سرویس ها و کم شدن مقدار مصرفی هر پاد

بعد از اینکه بار کمتر شد؛ این خودکار ساز شروع به کمتر کردن تعداد پاد ها بر اساس مرور زمان میکند. و همانطور که گفته شد حدودا ۵ دقیقه طول میکشد تا تعداد پاد ها را کم کند.



و همین طور که بار کمتر میشود؛ این خودکار ساز تعداد پاد ها را هم کمتر میکند تا منابع سخت افزاری ما بهینه مصرف شوند و اتلاف منابع نداشته باشیم.



شکل ۲۶ - کم کردن سرویس ها توسط خودکار ساز بعد از کم شدن بار

برای معیار حافظه ای هم به همین صورت عمل میکند و وقتی پاد ما از حدی که مشخص کردیم بیشتر حافظه مصرف کرد؛ تعداد پاد ها را افزایش میدهد تا این میزان مصرفی برای هر پاد کمتر شود. تفاوتش با معیار محاسباتی در این است که برای افزایش میزان مصرفی یک سرویس از تولید کننده بار siege استفاده نمی کنیم و در عوض از ابزار stress استفاده میکنیم که با استفاده از این ابزار می توانیم میزان مصرفی حافظه برای یک سرویس را افزایش دهیم. برای اینکار ابتدا باید وارد کانتینر شویم که با استفاده از دستور زیر این کار را انجام میدهیم.

Kubectl exec -it [pouya-nginx-ingress-controller-59f4fdb9dd-sffsd](http://localhost:8001/api/v1/namespaces/kubernetes-dashboard/services/https:kubernetes-dashboard:/proxy/#/pod/default/pouya-nginx-ingress-controller-59f4fdb9dd-sffsd?namespace=default) -- sh

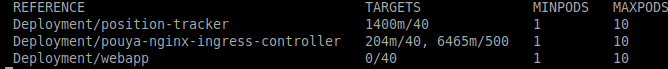
سپس بعد از نصب ابزار stress در کانتینر خود؛ از طریق دستور زیر میزان مصرفی حافظه را افزایش میدهیم.

stress --vm 2 --vm-bytes 200M

در این دستور ما مشخص میکنیم که دو thread هر کدام به میزان ۲۰۰ مگابایت شروع به مصرف حافظه بکنند. هنگامی که میزان مصرفی از حد تعیین شده رد شد؛ خودکار ساز ما پاد ها مثل معیار محاسباتی افزایش میدهد و بعد از تمام شدن بار؛ پاد ها را کاهش میدهد.

2 نتایج خودکار ساز برای custom metrics

حال نتایج این خودکار ساز را برای یک سرویس خود مورد بررسی قرار میدهیم. این سرویس همان وب سرور ما است به نام pouya-nginx--ingress-controller که درخواست ها را بین سرویس های مختلف ما پخش میکند. این سرویس نقش مهمی را در سیستم ما ایفا میکند و اگر درست کار نکند ؛ کاربران ما نمیتوانند به بقیه سرویس های ما دسترسی پیدا کنند. پس ما باید اطمینان حاصل کنیم که تاخیر این سرویس در حد قابل قبولی است و کار خود را به موقع انجام میدهد. حال وضعیت این سرویس را قبل از اعمال بار نگاه میکنیم که به صورت زیر است.

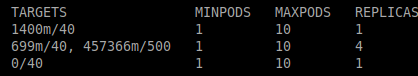
شکل ۲۷ - شمای کلی خودکار ساز

حال با بار پنج کاربر همزمان که درخواست می فرستند شروع میکنیم که با استفاده از load generator ای که در بخش قبل استفاده کردیم این بار را تولید میکنیم.

ابتدا یک توضیحی در مورد اعداد این خودکار ساز بدهیم که عدد ۴۰ در روبروی سرویس ما به این معنی است که اگر از ۴۰ درخواست بر ثانیه بیشتر شد؛ این خودکار ساز پاد ها را افزایش دهد. و عدد ۲۰۴m هم یعنی ۲۰۴mili requests که برابر ۰.۲ درخواست بر ثانیه است. پس یک درخواست بر ثانیه برابر ۱۰۰۰m است. همچنین کوبرنتیز برای اینکه با اعداد اعشاری کار نکند اعداد را در هزار ضرب میکند. عددی که در کنار عدد ۵۰۰ نوشته شده است را باید بر ۱۰۰۰ تقسیم کنیم که برابر ۶ میلی ثانیه است.

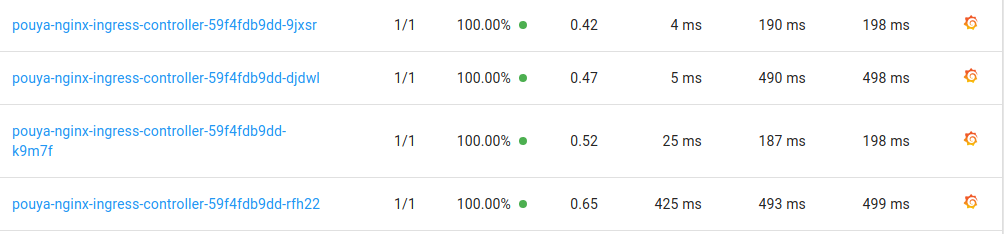
شکل ۲۸ - نتیجه خودکار ساز بعد از اعمال بار

بعد از اعمال این بار میبینیم که میزان پاسخ دهی بیشتر از حد مجاز شده است و باید این خودکار ساز ما تعداد پاد ها بیشتر کند تا این تاخیر کمتر شود.



شکل ۲۹ - نتیجه خودکار ساز بعد از مقیاس پذیری سرویس

همان طور که میبینیم این خودکار ساز تعداد پاد ها را افزایش داده و میزان تاخیر کمتر از حد مجاز شده است. در شکل عملکرد هر پاد را جداگانه مشاهده میکنیم.



شکل۳۰ - مربوط به نتایج Linkerd

میزان تاخیرهای P95P ,99 و P50 را برای این پاد ها در شکل بالا امده است و همچنین چه میزان خطا در پاسخ دهی داشتند که در این شکل خطایی در پاسخ دهی نبوده و وضعیت پاسخ دهی ۱۰۰ درصد است. همچنین تعداد درخواست برای هر پاد مشخص است که در ردیف چهارم از سمت راست آمده است.

2-1 انجام تست ها بر روی سرویس مد نظر:

حال بعد از آنکه نحوه عملکرد این خودکار ساز را مشاهده کردیم؛ سراغ انجام سناریو های مربوط به اعمال بار بر روی سرویس خود میرویم که به ترتیب این بار ها را با تعداد کاربر ۳۰۰؛ ۴۰۰؛ ۵۰۰ و ۷۰۰ تست میکنیم و ماکزیمم پاد ها را هم به ۱؛ ۳؛ ۶ و ۱۰ محدود میکنیم. زمان انجام تست ها برای همه ی اجرا ها ۴ دقیقه است.

الف) بار با ۳۰۰ کاربر:

بعد از اعمال بار با ۳۰۰ کاربر با استفاده از تولید کننده بار برای ماکزیمم پاد های مختلف؛ نتایج گوناگونی را استخراج کردیم که به صورت جدول در شکل زیر آمده است.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Max pods =10 | Max pods = 6 | Max pods = 3 | Max pods = 1 |  |
| 51139 | 50642 | 41892 hits | 27954 hits | Transactions |
| 99.76% | 99.78% | 99.72% | 99.53% | Availability |
| 1.37 secs | 1.39 secs | 1.68 secs | 2.52 secs | Response Time |
| 213 trans/sec | 211 trans/sec | 174 trans/sec | 116.74 trans/sec | Transaction Rate |
| 293.13 | 293.42 | 294.71 | 294.18 | Concurrency |
| 8.10 | 7.94 | 28.46 | 32.07 | Longest Transaction |

جدول۱ - نتایج تولید کننده بار برای ۳۰۰ کاربر

ب) بار با ۴۰۰ کاربر:

حال نتایج را برای این تعداد کاربر در جدول زیر مشاهده میکنیم.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Max pods =10 | Max pods = 6 | Max pods = 3 | Max pods = 1 |  |
| 56838 | 55394 | 42354 hits | 29059 hits | Transactions |
| 99.51% | 99.53% | 99.42% | 99.30% | Availability |
| 1.65 secs | 1.69 secs | 2.22 secs | 3.23 secs | Response Time |
| 236.87 trans/sec | 231.48 trans/sec | 177.07 trans/sec | 121.35 trans/sec | Transaction Rate |
| 389.93 | 390.98 | 392.51 | 391.52 | Concurrency |
| 16.11 | 22.01 | 43.65 | 52.59 | Longest Transaction |

جدول۲ - نتایج تولید کننده بار برای ۴۰۰ کاربر

ج) بار با ۵۰۰ کاربر:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Max pods =10 | Max pods = 6 | Max pods = 3 | Max pods = 1 |  |
| 58167 hits | 56160 hits | 42422 hits | 28115 hits | Transactions |
| 99.40% | 99.35% | 99.17% | 98.66% | Availability |
| 2.01 secs | 2.08 secs | 2.77 secs | 4.16 secs | Response Time |
| 243.25 trans/sec | 234.66 trans/sec | 177.04 trans/sec | 117.50 trans/sec | Transaction Rate |
| 488.10 | 488.48 | 489.52 | 488.90 | Concurrency |
| 42.30 | 46.28 | 55.58 | 75.58 | Longest Transaction |

جدول۳ - نتایج تولید کننده بار برا ۵۰۰ کاربر

د) بار با ۷۰۰ کاربر:

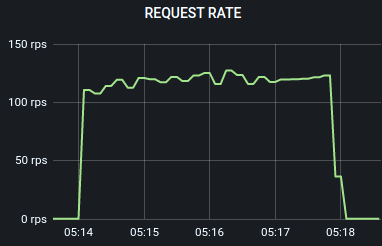
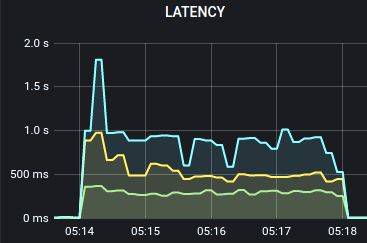
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Max pods =10 | Max pods = 6 | Max pods = 3 | Max pods = 1 |  |
| 57457 hits | 57885 hits | 42071 hits | 26705 hits | Transactions |
| 98.95% | 99.01% | 98.03% | 96.74% | Availability |
| 2.84 secs | 2.80 secs | 3.87 secs | 5.97 secs | Response Time |
| 239.59 trans/sec | 241.54 trans/sec | 175.32 trans/sec | 111.45 trans/sec | Transaction Rate |
| 679.88 | 675.66 | 678.45 | 665.38 | Concurrency |
| 91.88 | 119.60 | 110.88 | 135.24 | Longest Transaction |

جدول۴ - نتایج تولید کننده بار برای ۷۰۰ کاربر

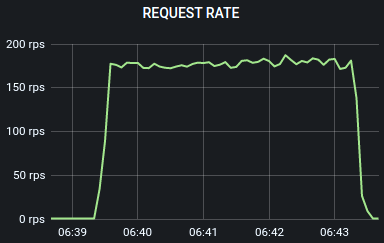
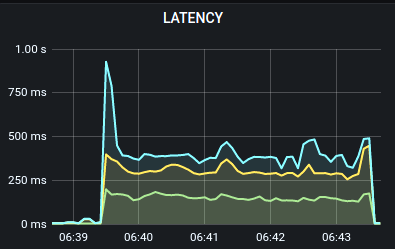
حال بعد از آنکه نتایج تولید کننده بار را مشاهده کردیم سراغ نتایجی که Linkerd به نشان داده است میرویم. سیستم Linkerd نتایج زمان پاسخ و تعداد درخواست در ثانیه را برای سرویس ما را با استفاده از Grafana به صورت شکل های زیر به نمایش میگذارد که در شکل های پایین این نتایج برای تعداد کاربر و ماکزیمم پادها نشان داده میشود. دلیلی که میزان زمان پاسخ برای تولید کننده بار و Linkerd متفاوت است؛ این است که زمان پاسخ در نتیجه تولید کننده بار؛مساوی جمع زمان پاسخ وب سرور ما و پاسخ سرویسی که وب سرور ما آن درخواست را به آن سرویس داده است. در صورتی که Linkerd تنها زمان پاسخ خود وب سرور ما را میدهد که همان سرویس nginx ما است.

الف) تعداد کاربر ۳۰۰:

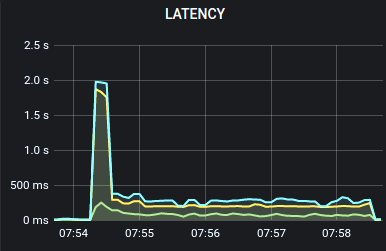
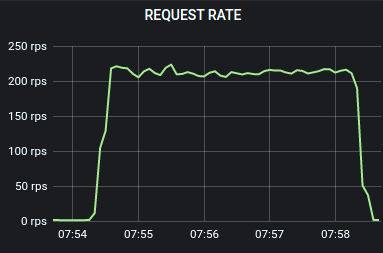
Max pods =1



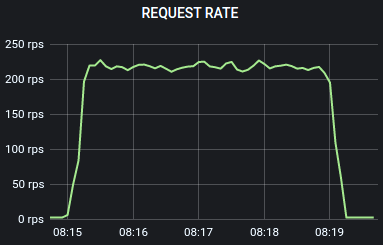
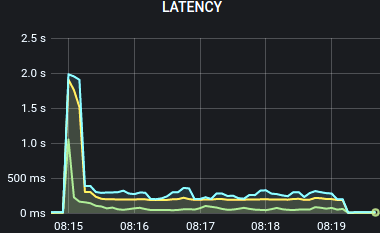
Max pods =3



Max pods = 6

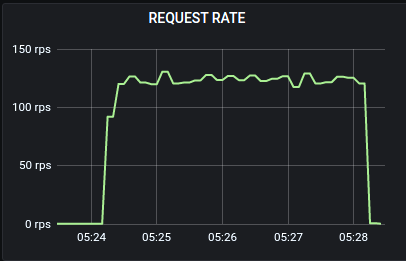
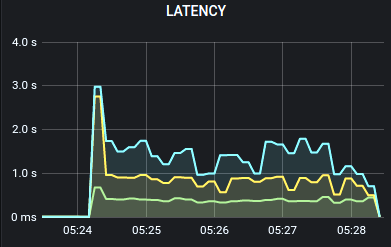


Max pods = 10

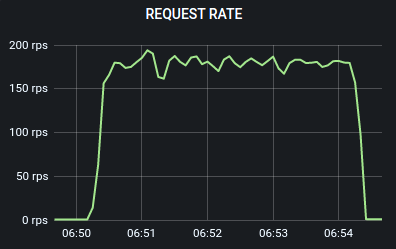
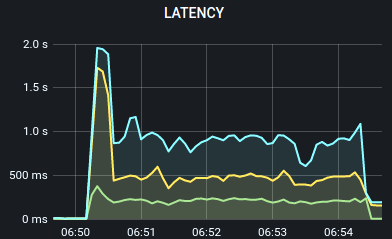


ب) تعداد کاربر ۴۰۰:

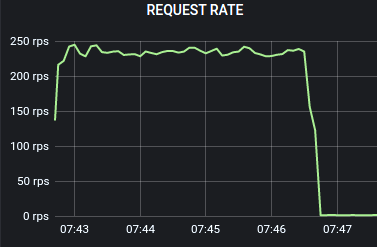
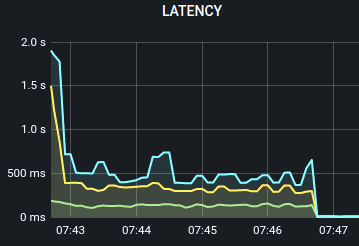
Max pods = 1



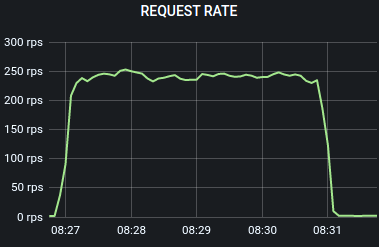
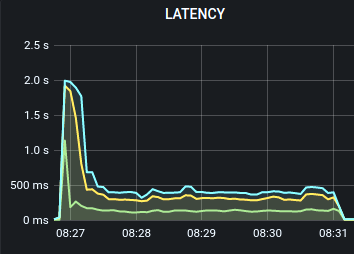
Max pods = 3



Max pods = 6

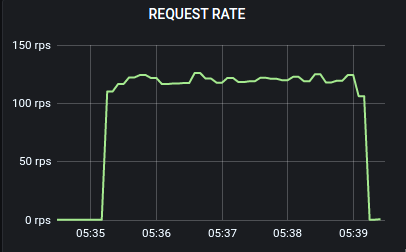
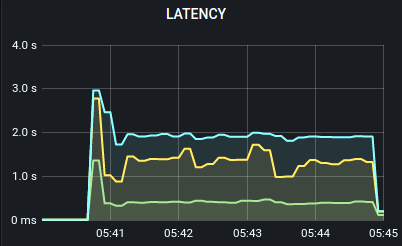


Max pods = 10

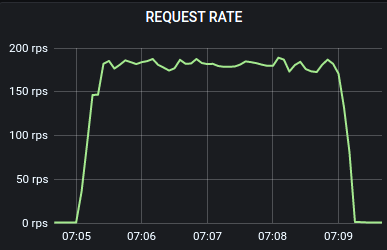
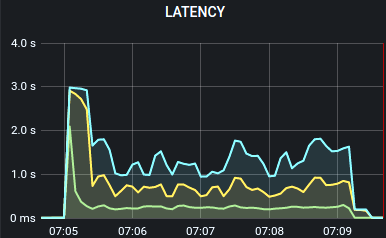


ج) تعداد کاربر ۵۰۰:

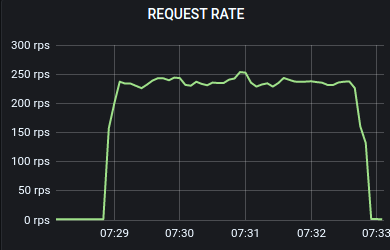
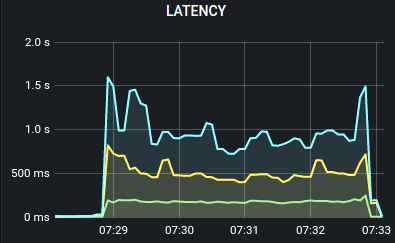
Max pods = 1



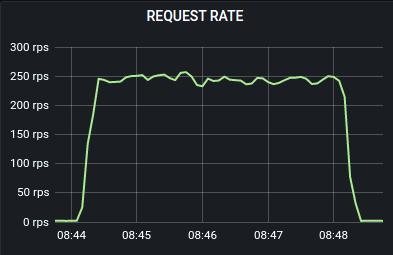
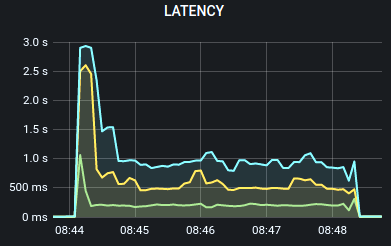
Max pods = 3



Max pods = 6

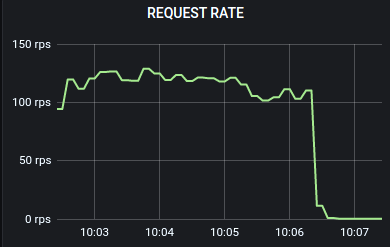
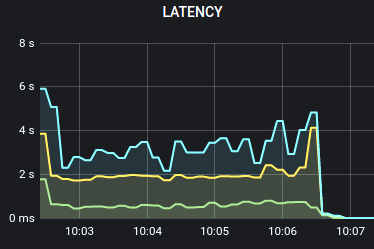


Max pods = 10

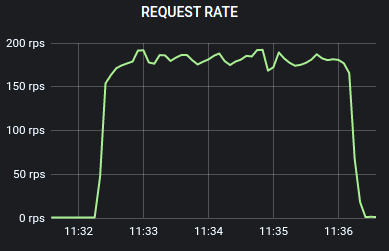
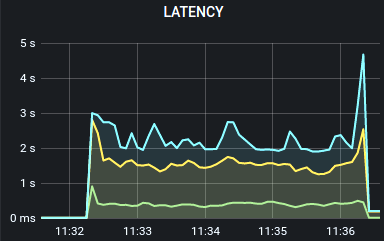


د) تعداد کاربر ۷۰۰:

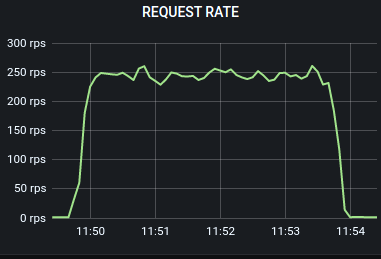
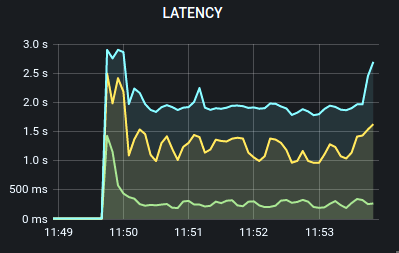
Max pods = 1



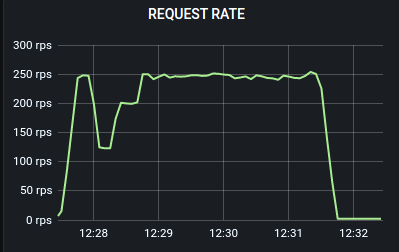
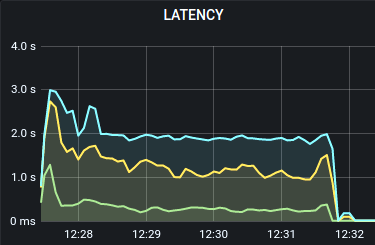
Max pods = 3



Max pods = 6



Max pods = 10



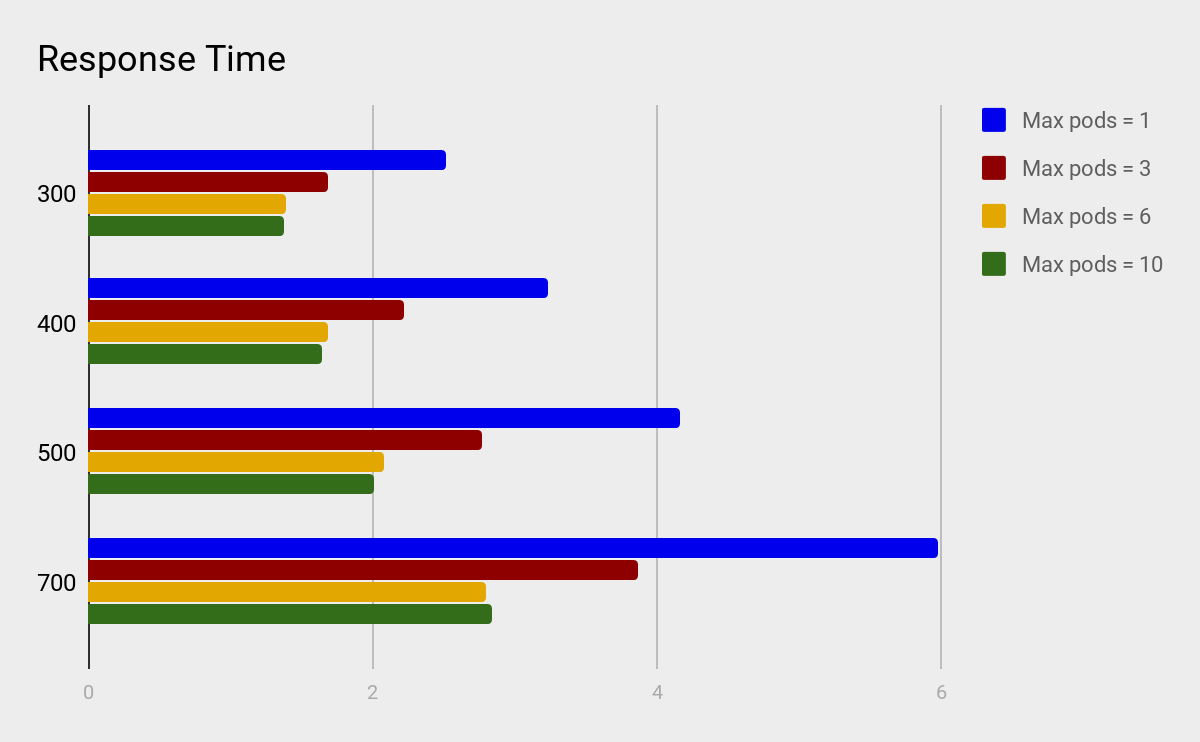
2-2 تحلیل نتایج:

بعد از آنکه نتایج را بدست آوردیم و گراف های مربوط به نتایج را هم مشاهده کردیم؛ سراغ تحلیل این نتایج میرویم. همان طور که در نتایج آمده است برای ۳۰۰ ؛ ۴۰۰؛ ۵۰۰ و ۷۰۰ کاربر تعداد ۳ پاد و ۶ پاد میزان زمان پاسخ را نسبت به یک پاد کمتر کرده است و این نشان میدهد که با تقسیم بار بر روی پاد ها؛ سرویس ما عملکرد بهتری دارد و زمان پاسخ کمتری نسبت به داشتن یک پاد نشان میدهد. ولی با توجه به اینکه منابع سخت افزاری ما محدود است این عملکرد تا یک جایی میتواند بهبود یابد و از یک حدی بیشتر دیگر افزایش پاد ها کمکی به بهبود زمان پاسخ ما نخواهد کرد و این به این منظور است که هر پاد از لحاظ نرم افزاری تعداد محدودی درخواست در ثانیه را می تواند جواب بدهد و هنگامی که بیش از حد توان خود بار دریافت می کند؛ خطا در جواب هم بالا میرود پس با تقسیم بار بر روی پاد ها می توانیم این مشکل را حل کنیم و باعث شود که هر پاد تعداد کمتری درخواست را پاسخ دهد. در آن طرف قضیه منابع سخت افزاری ما محدود است و هر پادی که تولید میشود یک بخشی از حافظه و CPU ما را اشغال میکند و اگر این منابع کم بیاید؛ سیستم ما کند میشود و دیگر قادر نخواهد بود به سرعت به درخواست ها جواب بدهد. پس اگر پاد ها را از یک حدی بیشتر کنیم؛ درست است که بار خیلی کمتر بر روی هر پاد می افتد و لی به دلیل آنکه منابع سخت افزاری هم محدود تر میشود؛ پاد های ما هم سرعتشان کمتر میشود. پس باید ما یک نقطه بهینه در سیستم خود پیدا کنیم که چه میزان پاد برای درخواست های ما مناسب است و برای تعداد درخواست هایی که ما در نظر گرفتیم ۶ پاد عملکرد سیستم ما رو بیشتر بهبود بخشیده و بیشتر از آن خیلی در نتایج فرقی ایجاد نمیکند.

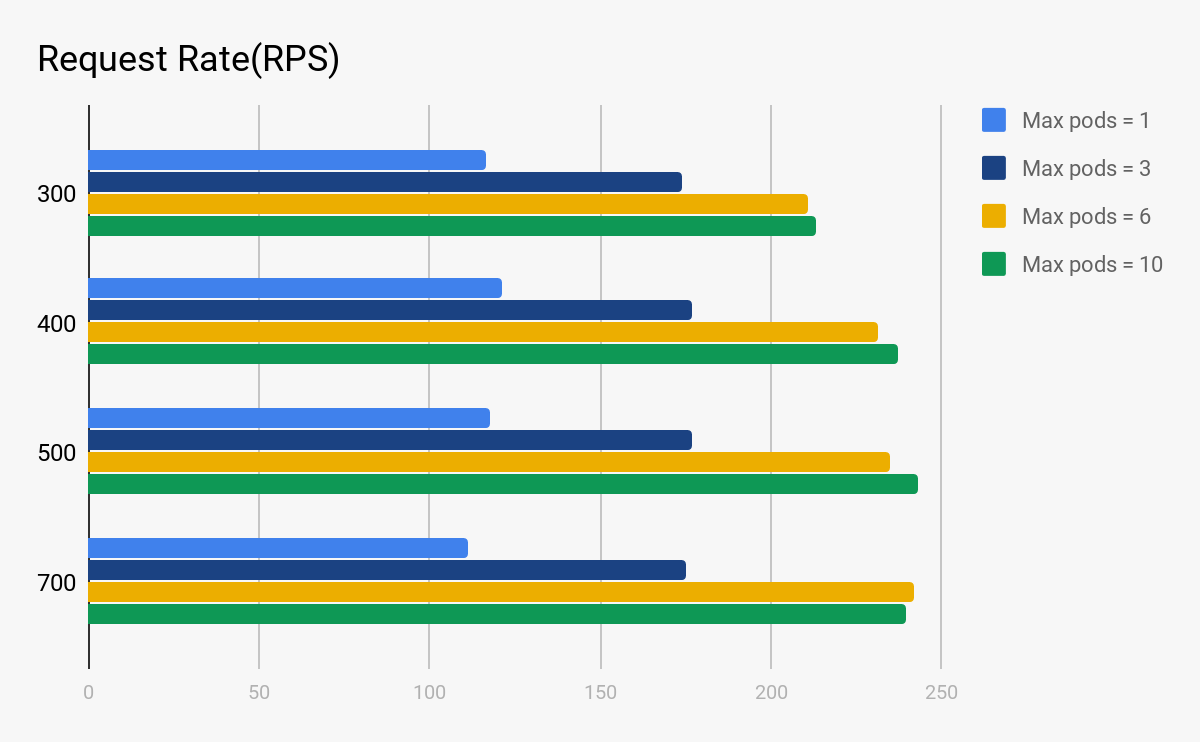
همچنین وقتی تعداد پاد ها را افزایش میدهیم و زمان پاسخ سرویس ما کمتر میشود به تبع آن میزان درخواست بیشتری در ثانیه را می تواند جواب دهد که این نتایج در گراف ها و جداول قسمت نتایج آمده است و همچنین جدولی برای مقایسه تعداد درخواست های مختلف و تاثیر افزایش پاد ها بر روی میزان تعداد درخواست در ثانیه را نشان میدهد. نکته ای که وجود دارد با توجه به مساله ای که در مورد مساله عملکرد نرم افزاری و محدودیت منابع سخت افزاری مطرح کردیم. با افزایش تعداد پاد ها تا یک حدی؛ عملکرد نرم افزاری ما بهتر میشود و پاد های ما بیشتر میتوانند پاسخگو باشند و تعداد بیشتری درخواست در ثانیه را جواب بدهند ولی بعد از یک حدی از تعداد پاد ها به دلیل محدودیت های سخت افزاری؛ سرویس ما نمی تواند تعداد بیشتری درخواست در ثانیه را جواب بدهد به دلیل آنکه دچار کمبود در منابع سخت افزاری می شویم و همین باعث میشود که تعداد بیشتر پاد ها به افزایش تعداد درخواست در ثانیه کمکی نکند.

همان طور که از نتایج ما مشخص است؛ با افزایش پادها میزان دسترس پذیری سیستم ما هم افزایش پیدا کرده است. درست است که این افزایش برای بار های پایین تر قابل توجه نیست ولی برای بار ۵۰۰ و ۷۰۰ این افزایش تا حدودی ملموس است و به دلیل آنکه بیشتر از ۷۰۰ سیستم دچار مشکل میشد تولید کننده بار ما قادر به ادامه نبود؛ امکان افزایش بار بیش از این وجود نداشت. در حالت کلی وقتی بار بر روی پاد ها تقسیم میشود و سرویس ما سریع تر بتواند درخواست ها را پاسخ بدهد و تعداد بیشتری درخواست پاسخ داده شود؛ به تبع آن دسترس پذیری هم افزایش میابد.

قسمت آخر از نتایج ما که میتواند قابل توجه باشد این است که با افزایش پاد ها حداکثر زمان پاسخ هم کم شده است. زمان پاسخی که در نتیجه تولید کننده بار آمده است متوسط همه زمان پاسخ ها است و بعضی از درخواست ها به دلیل آنکه بر روی سرویس ما بار زیادی افتاده است؛ طول میکشد پاسخ داده شود. ولی هنگامی که تعداد پاد ها را افزایش میدهیم و بار بین پاد ها تقسیم میشود؛ مشاهده میکنیم که حداکثر زمان پاسخ هم کاهش میابد و باعث بهبود عملکرد سرویس ما میشود.

****

شکل ۳۱ - مقایسه بارهای مختلف و ماکزیمم پاد ها برای زمان پاسخ



شکل ۳۲ - مقایسه بارهای مختلف و ماکزیمم پاد ها برای تعداد درخواست در ثانیه

3 جمع بندی:

در این فصل ما نتایج قیاس پذیری برای معیار های محاسباتی و حافظه ای را بدست آوردیم. همچنین نحوه عملکرد خودکار ساز برای این معیار ها هم نشان داده شد. در قسمت دوم مقیاس پذیری بر اساس معیار های زمان پاسخ و تعداد درخواست در ثانیه انجام شد. به دلیل آنکه این دو معیار در قسمت دوم اهمیت بیشتری دارند و عملکرد سرویس ها را بهتر و کاربردی تر نشان میدهند؛ ما سناریو های مختلفی برای مقیاس پذیری سرویس مد نظر خود در نظر گرفتیم و نتایج آن را به صورت گراف و جدول به نمایش گذاشتیم.