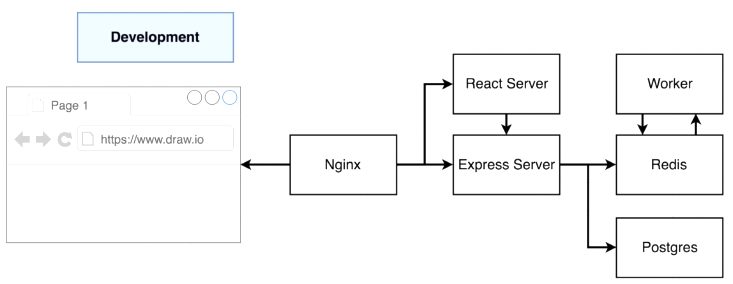
فصل سوم : توضیح سرویس ها و ایمیج کردن مایکرو سرویس ها با استفاده از سیستم داکر

۱. 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.

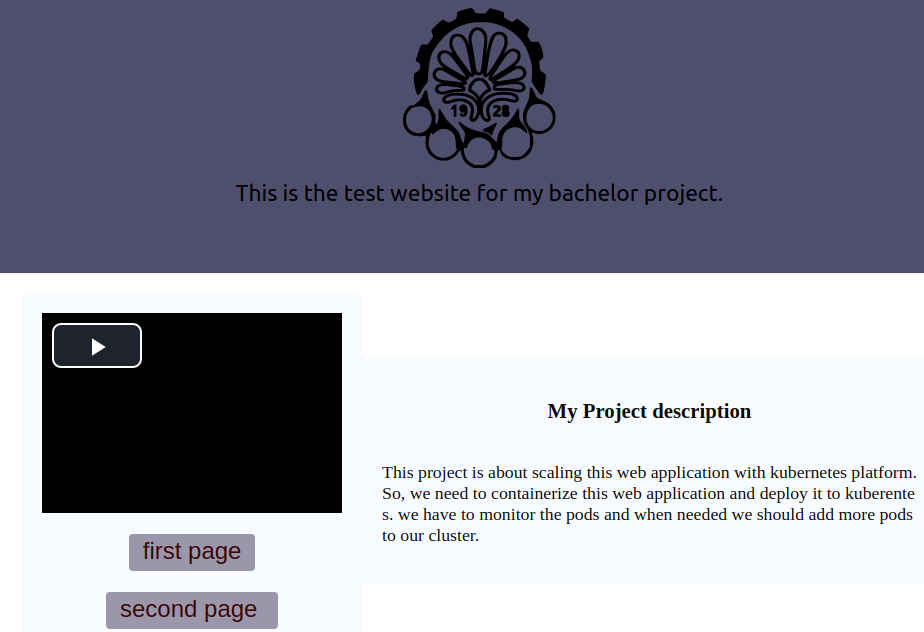
برای این پروژه ما یک سرویس کوچک و ساده را پیاده سازی میکنیم که پیچیدگی سیستم زیاد نشود و بیشتر بتوانیم در ابعاد مدیریت و مقیاس پذیری این سرویس ها کار کنیم. برای این منظور سرویس محاسبه فیبوناچی را مد نظر گرفته ایم که این سرویس را به چند مایکرو سرویس تقسیم بندی میکنیم. سیستم خود را به شیش مایکرو سرویس 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.



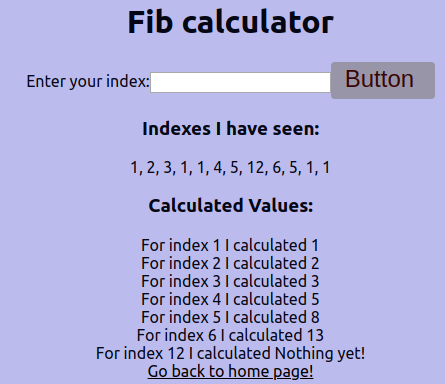
شکل ۱ - نحوه ارتباطات مایکرو سرویس ها با یکدیگر

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

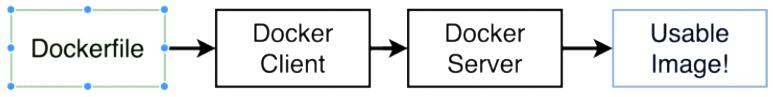


شکل ۲ - نتایج مایکرو سرویس های server؛ client؛ worker؛redis و postgres

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)

۱.۲. 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:

۱. مشخص کردن base image

۲. اجرا کردن دستوراتی برای نصب برنامه هایی که در کانتینر میخواهیم استفاده کنیم

۳. 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

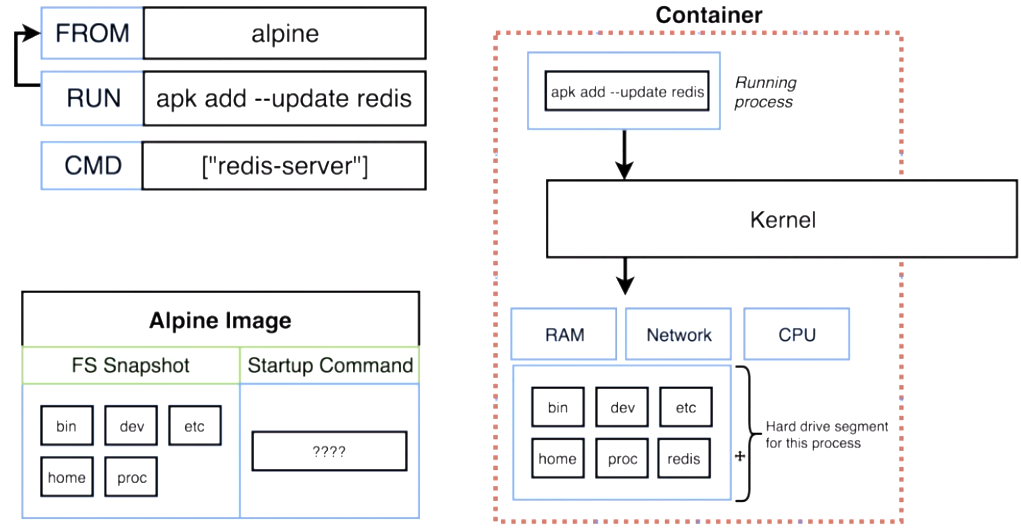
داکر سرور با خواندن این کانفیگ فایل دستوراتی که برای درست کردن یک ایمیج لازم است را میگیرد و ایمیج مد نظر ما را میسازد. دستورات دیگری هم برای dockerfile وجود دارد ولی این دستورات؛ دستورات کلی و مهم برای ساختن یک ایمیج هستند.

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.

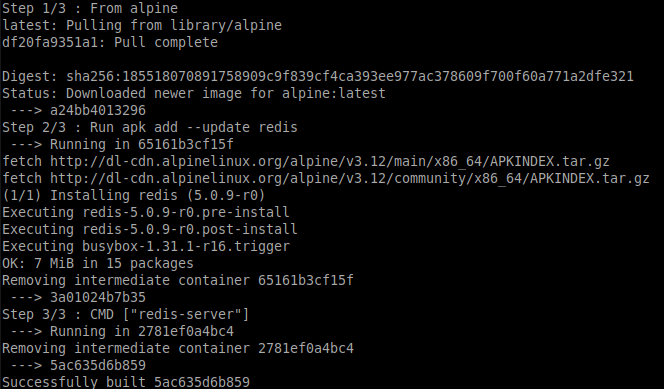
بعد از این توضیحات سراغ ساختن ایمیج مد نظر میرویم که میتوانیم با استفاده از دستور زیر این کار را انجام دهیم . " **.** docker build "

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.

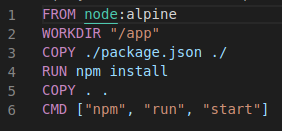
Docker build -t mpouaykh/redis:v2 **.**

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.



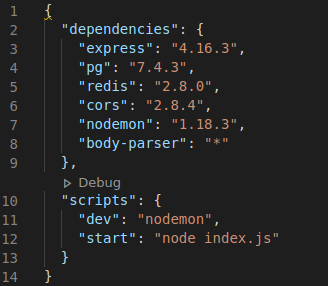
شکل۶ - dockerfile سرویس های server؛ worker و client

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.

۱. 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

۲. 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

۳. 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.

۴. 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

۵. swap را باید 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.

۶. 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.

۷. 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

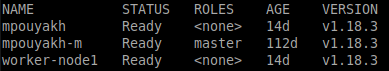
curl -sSL "https://github.com/coreos/flannel/blob/master/Documentation/kube-flannel.yml?raw=true" | sed "s/amd64/${ARCH}/g" | kubectl create -f -

۷. 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

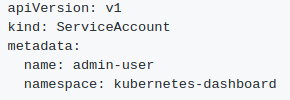
همچنین ما میتوانیم با استفاده از داشبورد کوبرنتیز که یک web UI است به کلاستر خود دسترسی پیدا کنیم و راحت تر بتوانیم کلاستر خود را مدیریت کنیم.

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

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

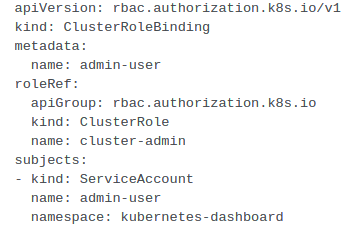
kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/v2.0.0/aio/deploy/recommended.yaml

۲. 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.



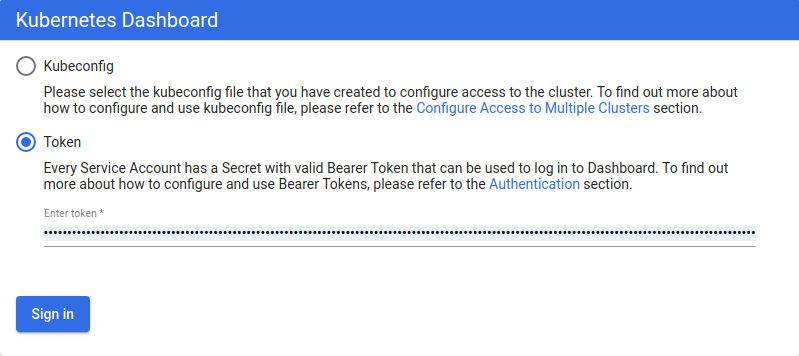
شکل ۳- اتصال کاربر جدید به نقش ادمین کلاستر

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

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

۱.۲. (RBAC (Role-Based Access Control

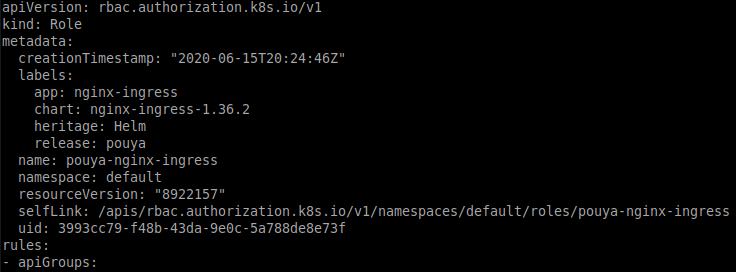
در کلاستر ما دو نوع کاربر داریم: یک کاربر حقیقی (User Accounts)که می تواند به کلاستر ما دسترسی پیدا کند و منابع را تغییر یا ایجاد کند و یک کاربر دیگر (Service Accounts) که برنامه ها و پروسه های درون کلاستر ما هستند که آنها هم میتوانند تغییراتی در کلاستر ما ایجاد کنند. به منظور

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.

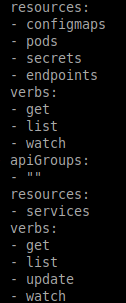
برای احراز هویت درخواست های سرویس ها هم باید از service account tokens استفاده کنیم که برای هر سرویس یک token به صورت آبجکت secret که باعث میشوند token های ما encode شوند و اطلاعات ما امن تر باشد؛ به این سرویس اختصاص می یابد. با استفاده از این token ها میتوانند با api server ارتباط برقرار بکنند.

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.



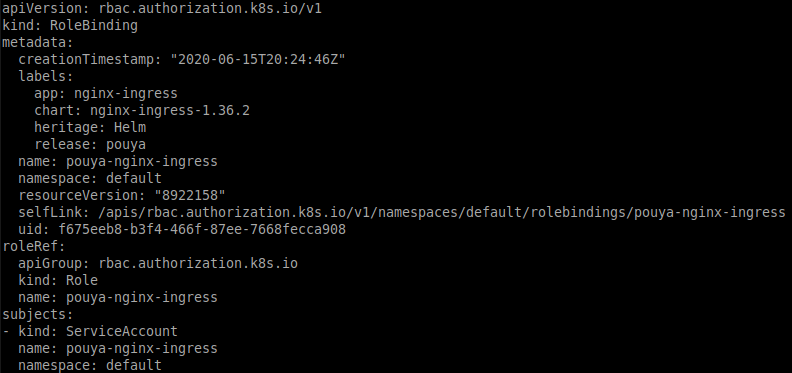
شکل ۵- آبجکت Role برای سرویس nginx

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.

۲.۲. 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.

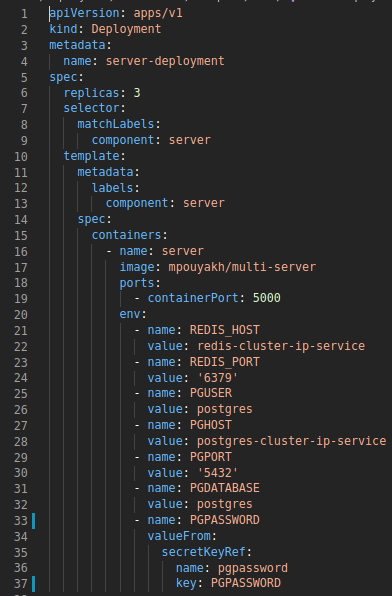
Deployment یک شیء سطح بالاست که به منظور تسهیل مدیریت چرخه‎ی حیات برای Podهای کپی شده در کلاستر طراحی شده است. پیکربندی Deployment را به راحتی می‎توان تغییر داد و تعداد Replica Set ها را تنظیم کرد. به دلیل وجود این ویژگی‏‎ها Deployment ابزار بسیار پرکاربردی است که در Kubernetes با آن اغلب کار می‎کنید.

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

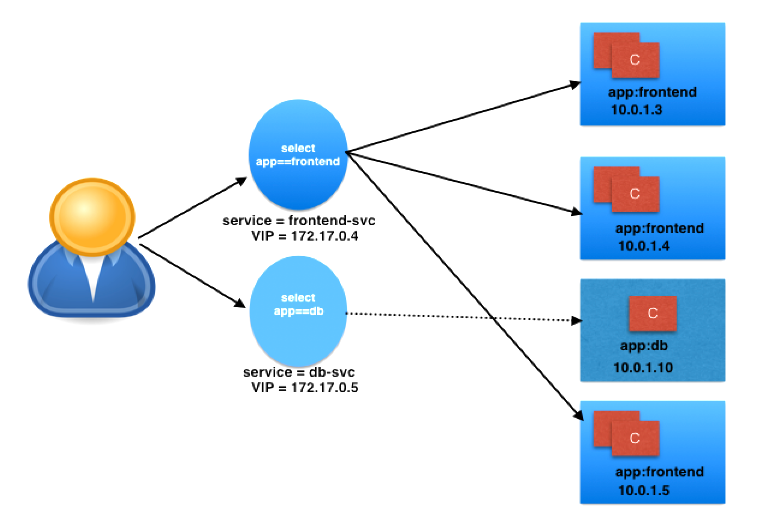
بخش form -literal-- به این منظور است که ما این آبجکت را با استفاده از دستور ترمینال اجرا میکنیم و نه به صورت نوشتن فایل و بعد بارگذاری آن در کوبرنتیز.

۳.۲. 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.

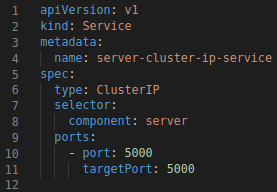


شکل ۹ - نحوه عملکرد آبجکت service در پیدا کردن پاد ها

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.

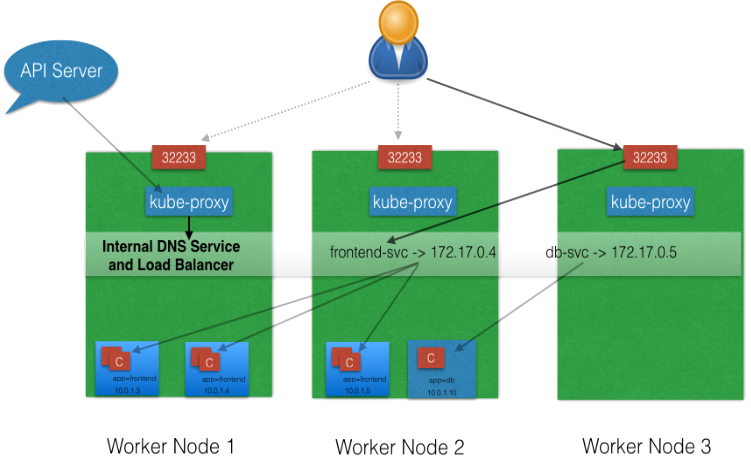


شکل ۱۰ - آبجکت service برای سرویس server

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.

۱.۳.۲. 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

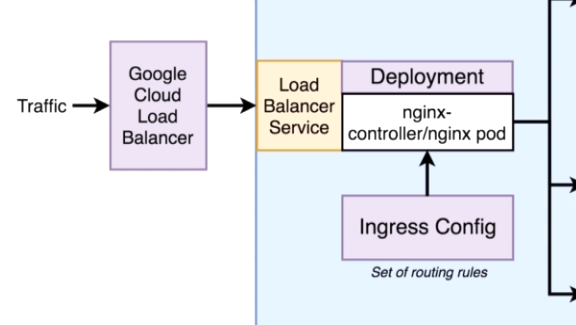
این نوع دسترسی از خارج کلاستر خیلی مطلوب نیست و بیشتر برای تست استفاده میشود. روش مناسب تری که میتوانیم برای آنکه از خارج کلاستر به پاد ها دسترسی پیدا کنیم؛ استفاده کنیم LoadBalancer است. این Loadbalancer به صورت خودکار ClusterIP و NodePort را برای اپلیکشین ما راه اندازی میکند و ترافیک های خارجی را از 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.

۲.۳.۲. 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.

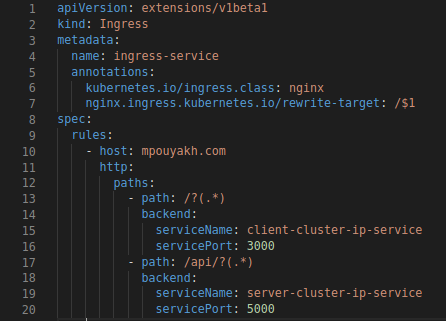


شکل ۱۲ - نحوه کارکرد سرویس LoadBalancer با Ingress

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.

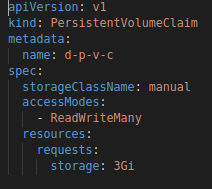
helm install pouya stable/nginx-ingress

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

۴.۲. 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.

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

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

۳. 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.

۱.۴.۲. Dynamic 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.

۱. sudo apt install nfs-kernel-server برای گره مستر و sudo apt install nfs-common برای گره های کارگر

۲. mkdir /svr/nfs/kubedata && sudo chown nobody: /svr/nfs/kubedata

۳. Sudo systemctl enable nfs-server && sudo systemctl start nfs-server

4. در فایل etc/exports/ خط زیر را وارد میکنیم.

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

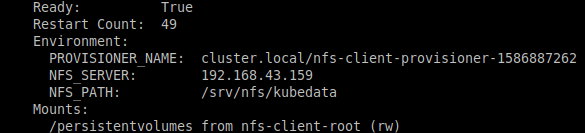
۵. sudo exportfs -rav

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.

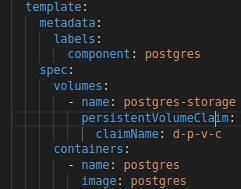
بعد از آنکه این مراحل را انجام دادیم و پاد nfs-provisioning در کلاستر ما اجرا میشود که با دستور kubectl pod describe وضعیت NFS-server و این پاد را مشاهده میکنیم.



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.

۳. 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.