RUNNING BITWARDEN_RS ON KUBERNETES

Jan. 3, 2021

Kyle Kaniecki

Intro

So as my girlfriend Emma and I setup the logistics of our new apartment in Boston, we quickly figured out that we had many accounts that would really be joint, but only filed under one person's name. Some of the bills were in my name, such as internet, while others were under her name like gas and electric, yet I felt that we both should have access to the accounts in case we needed to get a bill paid quickly.

Also, being two security minded people, Emma and I both use Bitwarden to create new passwords on the sites we visit so we have strong, unique passwords to our accounts. This makes it so that if one company has a data breach, not all of your accounts are leaked. However, Bitwarden only offers password sharing between organizations, which is a premium feature for the hosted service. However, since Bitwarden is open-source, you can self host the premium version on your own hardware if you would like and unlock all of the features. Since I was already paying for the kubernetes cluster that is hosting this blog, I figured I would throw a bitwarden server behind my traefik instance and utilize the nodes a little more efficiently. Two birds with one stone.

Research

So initially, while looking up bitwarden self hosted options, I first landed on the official Bitwarden Github Page. It looks like it could be run with Docker, which means that the image is hosted somewhere, which is perfect for what I needed!

However, as I started to do more testing locally with the docker image, the official image seemed to be very resource heavy... Ok, not *that* resource heavy, but certainly more than I wanted since it required an instanced of sql server to be running. So, I started to look for other open source projects around the web and stumbled upon bitwarden_rs. It is the Bitwarden API written in rust and it's rocket server. This was *exactly* what I was looking for. A lightweight, simple Bitwarden server that could write to small sqlite3 instanced inside the container in the /data directory.

I also really loved that Bitwarden *unofficially* supports bitwarden_rs and actually files issues in their github if problems arise. On the flip side, bitwarden_rs actively encourages their users to contribute to upstream development, whether it be through financial contributions or fixing bugs. So if you are following along with this blog, please consider donating to this wonderful service.

Implementation

The first thing I did before deciding on bitwarden_rs was read over their entire wiki which you can find here. If I miss anything in this blog article, I guarantee you it will be in their wiki, and I encourage my readers to also read the wiki before following along. It will help you understand my implementation further, and maybe even find things I have missed understand my implementation further, and maybe even find things I have missed.

Next, I dove into creating the kubernetes manifests. Since bitwarden_rs maintains a public docker image, it was pretty easy to get the Kubernetes deployment going. You can find the code below, with short explanations after each snippet:

```
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: bitwarden
  labels:
    app: bitwarden
data:
  SMTP_HOST: "smtp.domain.tld"
  SMTP_FROM: "user@smtp.domain.tld"
  SMTP_PORT: "587"
  SMTP_SSL: "true"
  # nginx-ingress-controller has built in support for Websockets
  # Project: https://github.com/kubernetes/ingress-nginx
  WEBSOCKET_ENABLED: "true"
  DATA_FOLDER: "/data"
  DOMAIN: "https://bitwarden.domain.tld"
  ROCKET_WORKERS: "5"
  SHOW_PASSWORD_HINT: "false"
  WEB_VAULT_ENABLED: "true"
  ROCKET_PORT: "8080"

  # Bitwarden RS settings
  SIGNUPS_ALLOWED: "false"
  LOG_FILE: "/data/bitwarden.log"
```

After reading over the wiki, it became apparent that bitwarden_rs uses environment variables for most of their user config. So, I collected all the variables I wanted to change from their defaults and stuck them in the above configmap. As you can see, I disable sign-ups and disable showing password hints to harden our server a little bit more, while also lowering the rocket worker number since I want the server to be super lightweight.

```
apiVersion: v1
kind: Secret
metadata:
  name: bitwarden-admin-token
  labels:
    app: bitwarden
type: Opaque
data:
  # openssl rand -base64 48
  token: ""
```

Next, the wiki recommended creating an admin token using openssl in order to be able to access the bitwarden admin page and create new organizations and such. However, in our case, we want to be able to store this secret in Kubernetes and attach it to our bitwarden service account (more on that next). So above is the yaml to create that secret in kubernetes and give it a name. I left a comment in there just so I don't forget how to create the token at a later date :)

```
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: bitwarden
  labels:
    app: bitwarden

---
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: Role
metadata:
  name: bitwarden
rules:
  - apiGroups:
      - ""
    resources:
      - configmaps
    resourceNames:
      - "bitwarden"
    verbs:
      - get
  - apiGroups:
      - ""
    resources:
      - secrets
    resourceNames:
      - "bitwarden-admin-token"
    verbs:
      - get

---
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: RoleBinding
metadata:
  name: bitwarden
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: bitwarden
subjects:
  - kind: ServiceAccount
    name: bitwarden
```

Next, I created a Kubernetes service account, role, and role binding to make sure that our bitwarden account that will be running our bitwarden container cannot perform CRUD operations arbitrarily on other resources in Kubernetes. Here, we specify in our role that our bitwarden service account can only access our secret we made above and the configmap that contains other non-secure configuration settings, and then bind it to the account.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: bitwarden
  labels:
    name: bitwarden
spec:
  selector:
    matchLabels:
      pod: bitwarden
  replicas: 1
  template:
    metadata:
      labels:
        pod: bitwarden
    spec:
      serviceAccountName: bitwarden
      containers:
        - name: bitwarden
          image: bitwardenrs/server:latest
          imagePullPolicy: IfNotPresent
          env:
            - name: ADMIN_TOKEN
              valueFrom:
                secretKeyRef:
                  name: bitwarden-admin-token
                  key: token
          envFrom:
            - configMapRef:
                name: bitwarden
          ports:
            - containerPort: 8080
              name: http
              protocol: TCP
            - containerPort: 3012
              name: websocket
              protocol: TCP
          resources:
            limits:
              cpu: 200m
              memory: 512Mi
            requests:
              cpu: 50m
              memory: 256Mi
          volumeMounts:
            - mountPath: /data
              name: bitwarden-data
  volumeClaimTemplates:
```

```
  - metadata:
      name: bitwarden-data
    spec:
      accessModes:
        - "ReadWriteOnce"
      resources:
        requests:
          storage: "5Gi"
      storageClassName: default
```

Ahem...

Ok, so there is a lot more going on here, so let's go over each part of this deployment. It is actually pretty straight forward if you are familiar with Kubernetes, but for completeness sake and when I inevitably forget this later, I will still review the important parts from the top of the document to the bottom:

- First, we give the deployment a common name (**bitwarden**) and attach a label to be able to query our deployment later.
- Next, we define the deployment specifications. I decided to only run one replica due to bitwarden_rs using sqlite3 inside the container and I didn't want to worry about concurrency with the sqlite file between replicas
- Next, in the template spec, I make sure that we are running our pods under our service account with our role that we created before
- I also define the containers that make up our pod, which happens to just be one: the bitwarden_rs official image
- Then, inside you can see I attach out config map to the container, as well as our token that we created in a secret above
- Open ports that are needed to communicate with the instance. 3012 is needed for websocket notifications
- Mount our volume that will hold our persistent sqlite3 file. This allows kubernetes to tear down our container if it crashes, but still save our data. Pretty cool.
- Then, I specify our volume templates. I only use a 5gi volume, as this is a lot of data for only a few family members or myself. But I imagine if you had more people uploading documents and other things, you could fill this up pretty quickly.

```
apiVersion: v1
kind: Service
metadata:
  name: bitwarden
spec:
  selector:
    pod: bitwarden
  type: ClusterIP
  ports:
    - name: http
      port: 80
      protocol: TCP
      targetPort: 8080
    - name: websocket
      protocol: TCP
      port: 3012
      targetPort: 3012
```

And then finally, we create a service that allows our bitwarden ports to be accessed by other pods in our cluster. Please note that I don't use the **loadbalancer** type for this service, as I am running Traefik in front of this server with Let's Encrypt, as suggested by the wiki.

Conclusion

Overall, this kubernetes deployment has been working well for me for about a week now. I have no problems accessing the server over the web when I am out of the house, and Emma and I have been sharing passwords across our house organization flawlessly for utility bills and other financials. It has really allowed us to be transparent about expenses as we move across the country to our new apartment in Boston!

As always, if you have an improvement to my blog, please feel free to leave a comment below! I am always trying to learn and grow.

devops kubernetes bitwarden service rust server kops aws deployment

COMMENTS

COMMENTS

Name

Email

Your email will not be displayed publicly

Comment

Please enter the letters into the box

Post