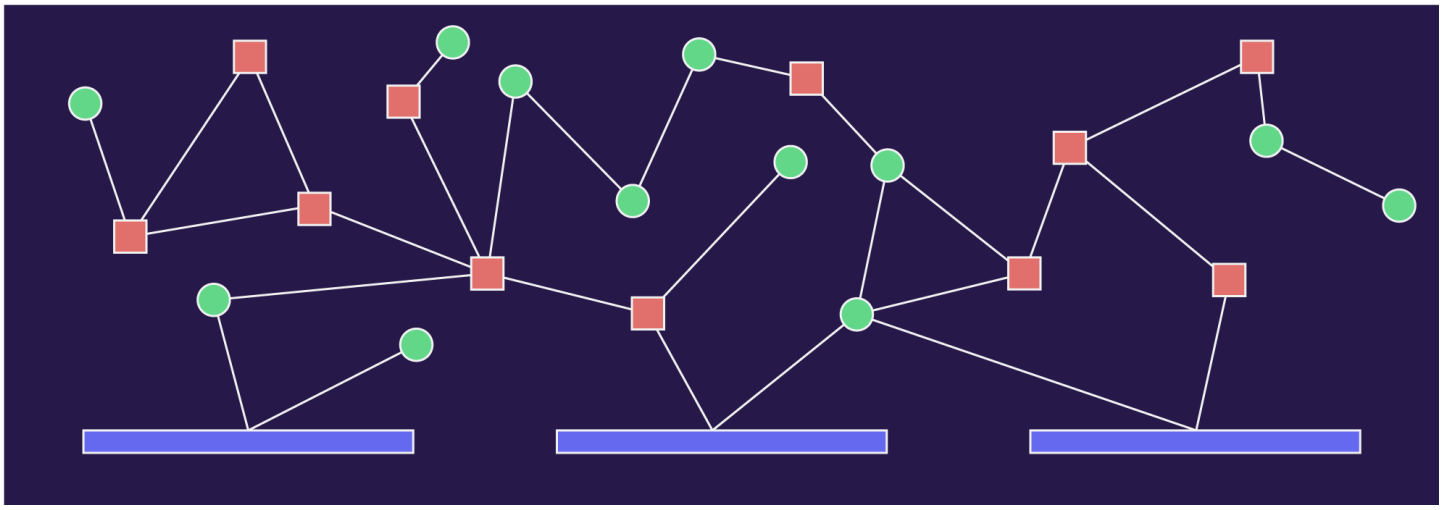
off bicycles.

Dec 24, 2017 · 8 min read

Getting started with microservices and Kubernetes



every microservices diagram ever

It's not a microservices platform if there's only one service. And all those services need to be able to talk to each other, they need to cope when some of them are not feeling well, they need to run on real machines, they need to be able to connect with the outside world and so much more besides.

This is where Kubernetes comes in—it orchestrates the life and times of individual Docker containers, giving us the primitives we need to construct robust and scalable systems.

These microservices things are kind of a big deal right now but there are few step by step guides to getting a basic system up and running. This is partly due to the fact that the notion of a *"basic microservice system"* is an oxymoron. We'll try regardless.

We do need some pre requisite knowledge, specifically what Docker is and what it's for. After that you'll need to know the Kube fundamentals: Pods, Services, Deployments et al.

This guide is mainly aimed at people who have got a single service running in Kube and are thinking "*now what?*".

Tldr; section

If you are more of a 'just show me the code' sort of person, you'll really like this git repo. Otherwise read on.

Before we start

All our microservices will be written in node.js v8.x so you'll want to go install that first. They'll all be very simple so you won't need more than the most cursory javascript / node knowledge.

We're going to run all this on Minikube, it's a neat way of getting Kube running locally. You can find installation instructions here. After that you'll want to verify that your Minikube installation is all good.

First create a Minikube cluster:

```
$ minikube start
```

```
Starting local Kubernetes v1.8.0 cluster...
Starting VM...
Getting VM IP address...
Moving files into cluster...
Setting up certs...
Connecting to cluster...
Setting up kubeconfig...
Starting cluster components...
Kubectl is now configured to use the cluster.
Loading cached images from config file.
```

Then check that the Kube system services are all happy:

```
$ kubectl get services -n kube-system
```

```
NAME                   CLUSTER-IP      EXTERNAL-IP   PORT(S)
AGE
kube-dns               10.96.0.10      <none>
53/UDP,53/TCP   1m
kubernetes-dashboard   10.107.19.167   <nodes>
80:30000/TCP    1m
```

One more thing, we need Minikube to share our local docker registry, else it won't be able to find the docker images that we build.

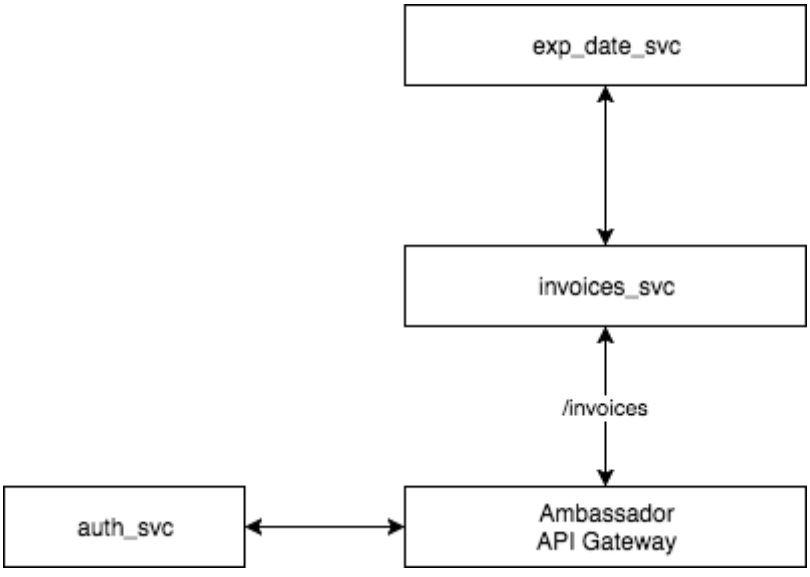
```
$ eval $(minikube docker-env)
```

Super. Now let's build something fun.

TOTAL INVOICE MANAGEMENT!!!1!

Lets build a system that manages invoices for a company. Sounds simple enough and it's also the most fun thing I could think of. Our system will comprise of:

- An **API gateway** to route traffic into our system
- An **authentication service** to limit access
- A front end **invoices service** to return information about invoices
- A back end **expected date service** that'll tell us when an invoice is likely to be paid



the basics

The first step is getting our folder structure sorted. We'll have one folder for all our kube config files, and others for each of our services.

```
- total_invoice_managment
|
| - kube
| - invoices_svc
```

The invoices service

Our first service is `invoices_svc` which is responsible for individual invoices. It'll have a single endpoint `api/invoices/:id` which will swap an id for the invoice data. Lets quickly scaffold the service using the node package manager (npm).

```
$ cd ./invoices_svc
$ npm init
# then say yes to everything
$ npm install express
```

Update `package.json` to include the script to boot the app:

```
{
  "name": "invoices_svc",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "node index.js"
  },
  "author": "",
```

Add the `index.js` file that contains the code for the service:

```
const express = require("express")

const app = express()

app.get("/api/invoices/:id", (req, res) => {
  const id = parseInt(req.params.id)
  res.json({
    id: id,
    ref: `INV-${id}`,
    amount: id * 100,
    balance: (id * 100) - 10,
    ccy: "GBP"
  })
})
```

Verify that it runs locally:

```
$ PORT=3007 npm start

invoices_svc listening on 3007

$ curl localhost:3007/api/invoices/10

{"id":10,"ref":"INV-
10","amount":1000,"balance":990,"ccy":"GBP"}
```

It works! Satisfied that our service works as expected, we can now dockerize it by making a Dockerfile:

```
FROM node:carbon
WORKDIR /usr/src/app

COPY package*.json ./

RUN npm install

COPY . .
```

Then we can build the Docker container to make sure all is well:

```
$ docker build ./ -t invoices_svc:v1
```

Time to start on getting this service into Kube. Lets change directory to the `kube` folder a level up:

```
$ cd ../kube
```

And add our first bit of kube config. Call the file `invoices_svc.yaml`

```yaml
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  labels:
    run: invoices-svc
  name: invoices-svc
  namespace: default
spec:
  replicas: 3
  selector:
    matchLabels:
      run: invoices-svc
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
    type: RollingUpdate
  template:
    metadata:
      labels:
        run: invoices-svc
    spec:
      containers:
      - image: invoices_svc:v1
        imagePullPolicy: IfNotPresent
        name: invoices-svc
        ports:
        - containerPort: 8080
      dnsPolicy: ClusterFirst
      restartPolicy: Always
```

This config defines a Kube service and it's accompanying deployment. We can ask kube to boot it up

```
$ kubectl apply -f ./invoices_svc.yaml
```

```
deployment "invoices-svc" created
service "invoices-svc" created
```

We should see its service:

```
$ kubectl get services
```

```
NAME           CLUSTER-IP      EXTERNAL-IP   PORT(S)   AGE
invoices-svc   10.104.86.220   <none>        80/TCP    3m
kubernetes     10.96.0.1       <none>        443/TCP   1h
```

And all the pods too:

```
$ kubectl get pods
```

```
NAME                            READY     STATUS    RESTARTS
AGE
invoices-svc-65b5f7bbd4-ckr8d   1/1       Running   0
44s
invoices-svc-65b5f7bbd4-gvk9s   1/1       Running   0
44s
invoices-svc-65b5f7bbd4-z2kx7   1/1       Running   0
44s
```

As there's no external IP for `invoices_svc` we'll need to get into a container *inside* the cluster to be able to try it out. Spinning one up specially seems odd, but it's a very kubey way of doing things. Busyboxplus is just a container that has a basic shell and some common tools. We need it to use `curl`.

```
$ kubectl run curl --image=radial/busyboxplus:curl -i --tty

[ root@curl-696777f579-qwjcr:/ ]$ curl
10.104.86.220/api/invoices/1
```

```
{"id":1,"ref":"INV-1","amount":100,"balance":90,"ccy":"GBP"}
```

(To escape the container you need to press `ctl-d)`

It works! Sort of. It's pretty useless being stuck inside our cluster - we need to create an *ingress* so that traffic can find it's way in. We are going to use Ambassador for this. It's a handy wrapper around Envoy Proxy and has lots of great API gateway features built in. Routing seems like a good place to start.

We'll need to get Ambassador running on our cluster. Create a file called `ambassador.yaml` in the `kube` folder:

```
---
apiVersion: v1
kind: Service
metadata:
  labels:
    service: ambassador-admin
  name: ambassador-admin
spec:
  type: NodePort
  ports:
  - name: ambassador-admin
    port: 8877
    targetPort: 8877
  selector:
    service: ambassador
---
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRole
metadata:
  name: ambassador
rules:
- apiGroups: [""]
  resources:
  - services
  verbs: ["get", "list", "watch"]
- apiGroups: [""]
  resources:
  - configmaps
  verbs: ["create", "update", "patch", "get", "list", "wat
- apiGroups: [""]
  resources:
  - secrets
  verbs: ["get", "list", "watch"]
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: ambassador
---
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: ambassador
roleRef:
```

```
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: ambassador
subjects:
- kind: ServiceAccount
  name: ambassador
  namespace: default
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: ambassador
spec:
  replicas: 3
  template:
    metadata:
      labels:
        service: ambassador
    spec:
      serviceAccountName: ambassador
      containers:
      - name: ambassador
        image: datawire/ambassador:0.19.2
        imagePullPolicy: Always
        resources:
          limits:
```

And then we can boot it up:

```
$ kubectl apply -f ./ambassador.yaml
```

```
$ kubectl get services
NAME               CLUSTER-IP       EXTERNAL-IP   PORT(S)
AGE
ambassador         10.103.215.136   <pending>
80:32005/TCP     11s
ambassador-admin   10.104.3.82      <nodes>
8877:31385/TCP   11s
invoices-svc       10.104.86.220    <none>        80/TCP
45m
kubernetes         10.96.0.1        <none>        443/TCP
2h
```

We need to tell ambassador about our `invoices_svc` though, and we do so by adding some annotations to the `Service` section of

`invoices_svc.yaml`

```
---
apiVersion: v1
kind: Service
metadata:
  labels:
    run: invoices-svc
  name: invoices-svc
  namespace: default
  annotations:
    getambassador.io/config: |
      ---
      apiVersion: ambassador/v0
      kind:  Mapping
      name:  fws-invoices_mapping
      prefix: /invoices/
      rewrite: /api/invoices/
      service: invoices-svc
```

The `prefix` key routes traffic from `/invoices/` to our service. To keep things nice and tidy the `rewrite` key does a bit of transforming too so that traffic to `/invoices/:id` gets routed to our service at `/api/invoices/:id.`

Once the config has been added, we can apply it:

```
$ kubectl apply -f ./invoices_svc.yaml
```

Ambassador keeps watch over everything that happens in the cluster. When we updated the config, ambassador detected that change and went looking for any annotations. It found them, and will now route traffic to the service.

In theory, we now have a working external api gateway to our cluster. Before we can validate that hypothesis we need to create a tunnel from our localhost to the minikube cluster:

```
$ minikube service ambassador --url
```

```
http://192.168.99.100:32005
```

This particular url is only for my local machine—**you need to use your own for future steps**.

We can use the returned url to reach our cluster:

```
$ curl http://192.168.99.100:32005/invoices/42
```

```
{"id":42,"ref":"INV-
42","amount":4200,"balance":4190,"ccy":"GBP"}
```

🎉 It works! So we have a service and a gateway.

Adding authentication

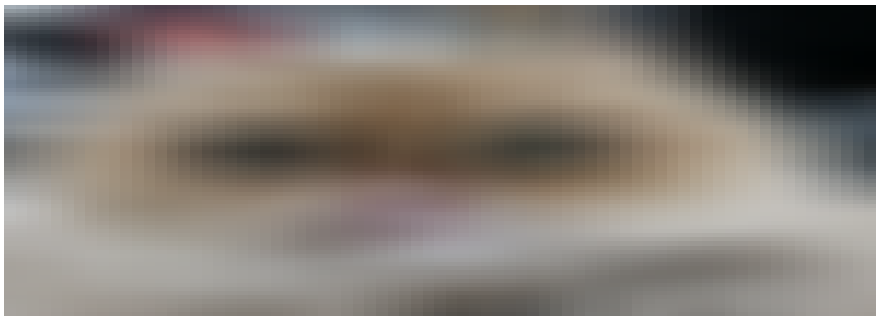
It's not great having our service available to world + dog. We should add some kind of authentication to our gateway. Nobody will be surprised to hear that we'll want a new service for that, or that it'll be called `auth_svc`.

- Create a new folder called `auth_svc`
- Copy the `Dockerfile` from `invoices_svc`
- Repeat the npm steps that we did for `invoices_svc`

```
$ cd ../
$ mkdir auth_svc
$ cd ./auth_svc
$ npm init
$ npm install express
$ cp ../invoices_svc/Dockerfile .
```

```
# don't forget to add "start": "node index.js" to your
package.json!
```

- Create the `auth_svc` app:



- Create the kube config:

- Build the docker image:

```
$ docker build -t auth_svc:v1 ./auth_svc/
```

- Apply the kube config:

```
$  kubectl apply -f ./kube/auth_svc.yaml
```

- see if it worked:

```
$ curl http://192.168.99.100:32005/invoices/42
{"ok":false}
```

Aces, we are now locked out, unless we know the magic word:

```
$  curl http://192.168.99.100:32005/invoices/42 -H
'authorization: letmeinpleasekthxbye'
```

```
{"id":42,"ref":"INV-
42","amount":4200,"balance":4190,"ccy":"GBP"}
```

Let's take stock. We have an API gateway that authenticates traffic and routes it to our service. However we don't want **all** of our services to be public, what about back end services that our front end services call? Well, Kube has a way of doing that too.

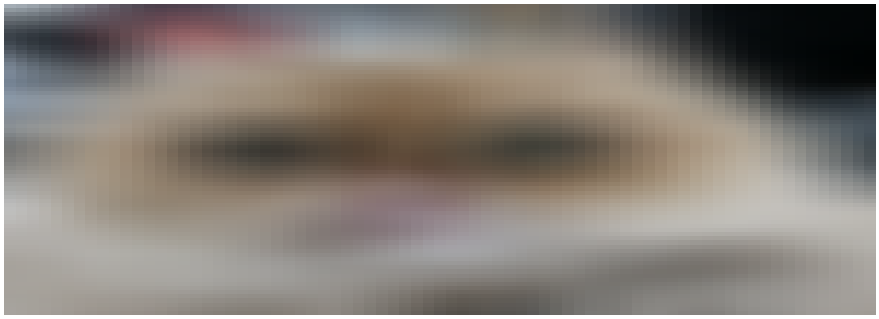
When do I get paid?

It's always nice to know when your customers will pay you. We will create an *extreme high sophistication algorithmic inference engine** that'll tell us when an invoice is expected to be paid. It's a similar jig to the last two services:

```
$ cd ../
$ mkdir expected_date_svc
$ cd ./expected_date_svc
$ npm init
$ npm install express
$ npm install moment
$ cp ../invoices_svc/Dockerfile .
```

```
# don't forget to add "start": "node index.js" to your
package.json!
```

And the *extreme high sophistication algorithmic inference engine* code is*:*



That just leaves the kube config:

You know the drill:

```
$ docker build -t expected_date_svc:v1 .
$ kubectl apply -f ../kube/expected_date_svc.yaml
$ kubectl get services
```

```
NAME                CLUSTER-IP       EXTERNAL-IP   PORT(S)
AGE
ambassador          10.103.215.136   <pending>
80:32005/TCP     19h
ambassador-admin    10.104.3.82      <nodes>
8877:31385/TCP   19h
auth-svc            10.108.119.134   <none>        3000/TCP
18h
expected-date-svc   10.101.227.50    <none>        80/TCP
1m
invoices-svc        10.104.86.220    <none>        80/TCP
20h
kubernetes          10.96.0.1        <none>        443/TCP
21h
```

So now we have the `expected_date_svc` running, we'll want to modify the `invoices_svc` to make us of it.

There's a new dependency we need to make a http request:

```
$ cd ../invoices_svc
$ npm install request-promise
$ npm install request
```

Then we make a request to the `expected_date_svc` and add the result to our invoice object. Here's the updated `invoice_svc` :

We need to rebuild the docker image:

```
$ docker build -t invoices_svc:v2 .
```

And we also need to update the kube config for the `invoices_svc`

First up, it needs to reference the new docker image:

We also need to add an environment variable that contains the url to the `expected_svc`. This is the nifty bit. Kubernetes uses internal DNS routing—you can read more about that here. The short version is that kube creates a special url for every named service. Its format is `SVCNAME.NAMESPACE.svc.cluster.local`, so the `expected_date_svc` can be found at `expected-date-svc.default.svc.cluster.local`. Lets go set that environment variable by updating the config:

Now that the config is all updated, we apply it to the cluster:

```
$ kubectl apply -f ../kube/invoices_svc.yaml
```

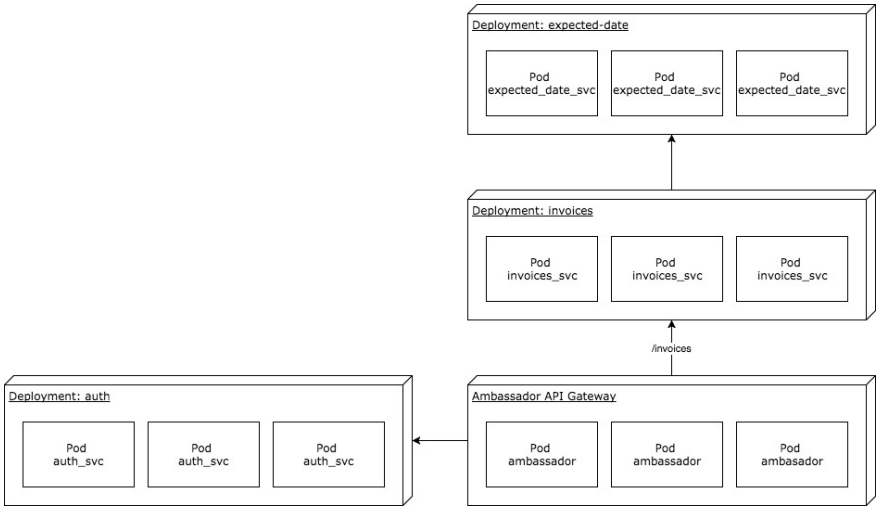
And check that the expected date is being added:

```
$ curl http://192.168.99.100:32005/invoices/42 -H
'authorization: letmeinpleasekthxbye'
```

```
{"id":42,"ref":"INV-
42","amount":4200,"balance":4190,"ccy":"GBP","expectedDate":
"2018-01-01T11:54:30.769Z"}
```

. . .

This should be enough for the reader to get a cluster running. Next steps include adding and removing replicas to scale services, adding a liveness probe so that kubernetes knows if a service fails silently or logging and monitoring so we can find out what our services are up to when we aren't looking.



How all the bits go togther

I like it!

Great, us too. We like kube so much that we use it for our most demanding infrastructure requirements at Fluidly, in particular for our

data science models. It's a steep learning curve, but the rewards are substantial.

If you like the sound of this sort of work we are often looking for amazing people. Drop us a line: jobs@fluidly.com .

. . .

* our data scientists do this for real!