Kubernetes in Depth
Lab 3
Working with networking

1 REFERENCES AND CHEAT SHEETS 1

2 COMMANDS COVERED 1

3 LAB SETUP 1

4 SETTING UP SAMPLE APP FOR COMMUNICATION 2

4.1 TESTING USERS AND TASKS APP 2

4.2 CREATING A DEPLOYMENT 7

5 INTRA-POD CONTAINER COMMUNICATION 11

6 CREATING DEPLOYMENTS FOR MULTIPLE PODS 18

7 INTERPOD COMMUNICATION 20

7.1 USING INTERNAL IP ADDRESS OF THE SERVICE 21

7.2 USING AUTOMATICALLY GENERATED ENVIRONMENT VARIABLES 24

7.3 USING COREDNS FOR INTERNAL DOMAIN NAMES OF SERVICES 28

8 ADDING THE TASK APP AS A DEPLOYMENT 30

9 END 36

1 References and cheat sheets

The official reference for Kubernetes networking:
https://kubernetes.io/docs/concepts/services-networking/

Additional references
https://spacelift.io/blog/kubernetes-networking
https://www.tigera.io/learn/guides/kubernetes-networking/

2 Commands covered

3 Lab setup

The root folders for all the various projects here can be found in the `Lab 3` subfolder in the main `labcode` folder of your downloaded zip for this workshop.

4 Setting up sample app for communication

The root folder for this project is: `kub-network-01-starting-setup`
Open some Powershell terminals in this folder to work with it.

The dummy Node.js application consists of three different backend APIs that work together, ultimately forming three separate containers. This application simulates the functionality that would typically be expected in a real world multi-tier full stack application.

The first API is the auth API, which handles verifying and generating tokens for authenticated users. Next is the users API, responsible for creating user accounts and logging them in. This functionality operates in a dummy mode, as no database or file storage is used. Instead, we work with temporary data since the focus here is to demonstrate network interaction between different services. The users API interacts with the auth API, for instance, to obtain tokens during the login process.

Finally, we have the tasks API, which can return a list of tasks or store new tasks through a simple form of file-based dummy storage for tasks. Similar to the users API, the tasks API also interacts with the auth API to verify tokens provided by logged-in users. These token would have been obtained earlier in the interaction between the users API and the auth API.

The three Node.js APIs are to be deployed as containers within pods within a cluster.

At the start, the auth API and the users API will each run in their own containers but will share the same pod. This setup allows for pod-internal communication, with the auth API accessible only from within the pod. The users API communicates with the auth API, particularly when a new user is created. For instance, when a request is sent to the users API to create a new user, the users API contacts the auth API to generate a token for that user.

Meanwhile, the tasks API will be deployed in a separate pod. Both the pods will be accessible through exposure from a service, however the auth API itself will not directly handle external requests—it will only be accessed through the users API. This is our initial setup, and we will refine it as we progress through this lab.

4.1 Testing users and tasks app

For now, we will start with an even simpler configuration: running the users API independently to handle incoming requests without communicating with the auth API.

We can first test this out with all 3 apps running in basic Docker containers:

```
docker compose up -d --build
```

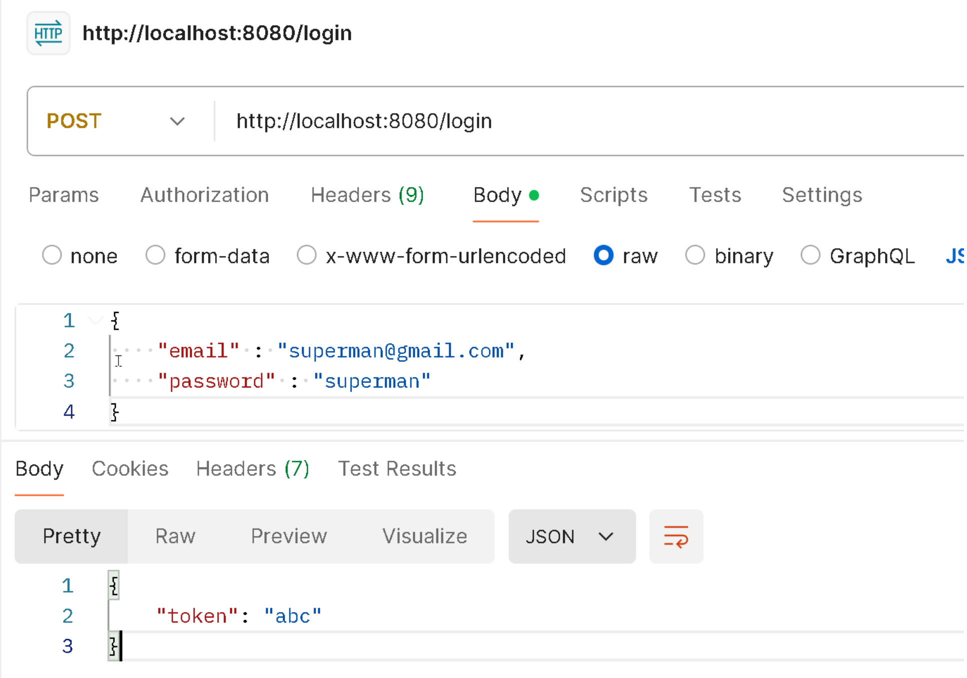
Check that all 3 services are up and running with:

```
docker compose ps
```

You should see 3 services (`auth, tasks` and `users`) up and running, with both `tasks` and `users` having their internal ports of 8080 and 8000 (where the application is listening on) being mapped to identical localhost ports.

Use REST client such as Postman to send a POST request to the `users` app that is currently listening on its container port 8080:
http://localhost:8080/login
with a dummy email/password pair in JSON format to receive back a dummy authorization token, for e.g.

```
{
    "email" : "superman@gmail.com",
    "password" : "superman"
}
```

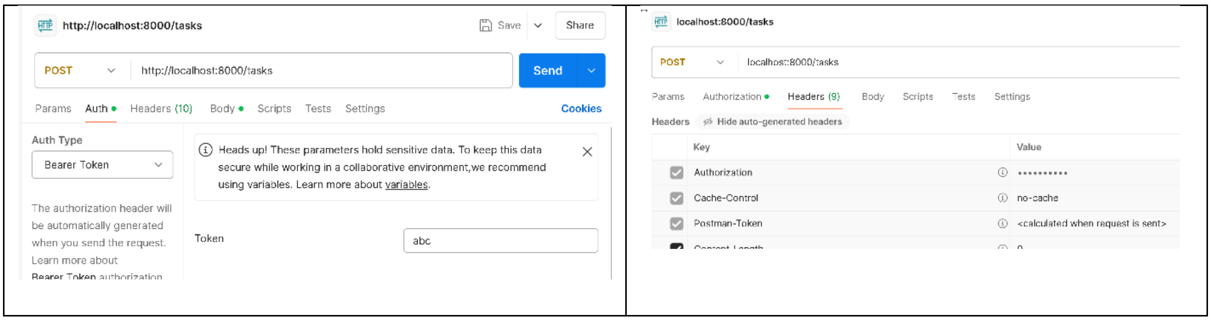


```
{
    "email" : "superman@gmail.com",
    "password" : "superman"
}
```

```
{
    "token": "abc"
}
```

You can then use the token returned to make API calls to create / retrieve tasks to the `tasks` app in the container listening at port 8000.

In order to do this, you will first need to configure the returned token (`abc`) as an Authorization token in an additional header to be sent out in all the HTTP requests sent out to this app. You can set this in the Authorization section of the Request page on Postman, using Auth Type as Bearer Token and setting `abc` as the token value. You should then be able to view this as an additional header in the Headers section.



http://localhost:8000/tasks

POST http://localhost:8000/tasks

Auth Type: Bearer Token

The authorization header will be automatically generated when you send the request. Learn more about Bearer Token authorization

Heads up! These parameters hold sensitive data. To keep this data secure while working in a collaborative environment,we recommend using variables. Learn more about variables.

Token: abc

localhost:8000/tasks

POST localhost:8000/tasks

Headers

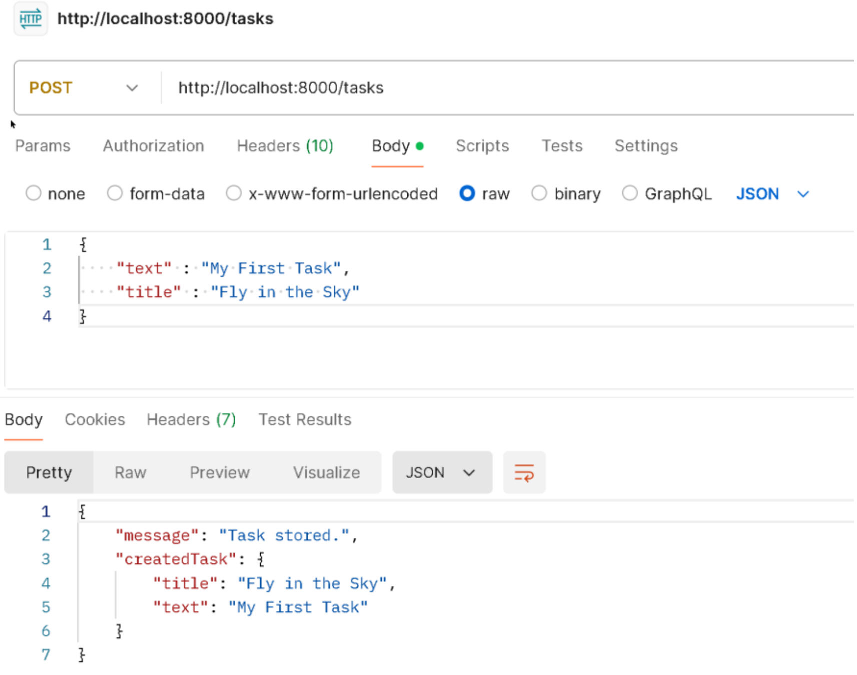
	Key	Value
☑	Authorization	••••••••••
☑	Cache-Control	no-cache
☑	Postman-Token	<calculated when request is sent>

Once this is done, you can configure the Body section some dummy data to be stored, for e.g.

```
{
    "text" : "My First Task",
    "title" : "Fly in the Sky"
}
```

and then click Send.

The app should return a response indicating successful retrieval of the token and storing of the specified task.



http://localhost:8000/tasks

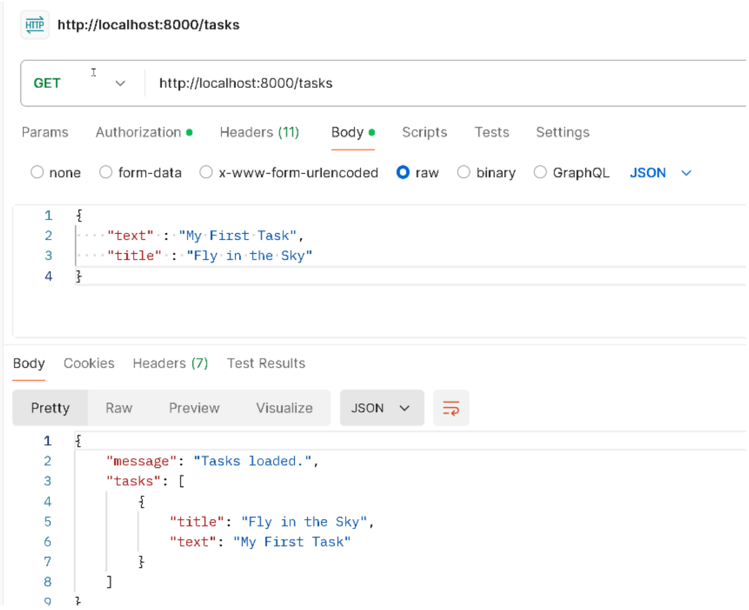
POST http://localhost:8000/tasks

○ none ○ form-data ○ x-www-form-urlencoded ● raw ○ binary ○ GraphQL JSON

```
{
    "text" : "My First Task",
    "title" : "Fly in the Sky"
}
```

```
{
    "message": "Task stored.",
    "createdTask": {
        "title": "Fly in the Sky",
        "text": "My First Task"
    }
}
```

Finally, you can send a GET request to the same API endpoint to retrieve the stored task. You need to do sent the GET request in the same tab in Postman, or else if you wish to do this in a new tab, you will need to configure the Authorization token of `abc` again.



GET http://localhost:8000/tasks

Body: ● raw JSON

```
{
····"text"·:·"My·First·Task",
····"title"·:·"Fly·in·the·Sky"
}
```

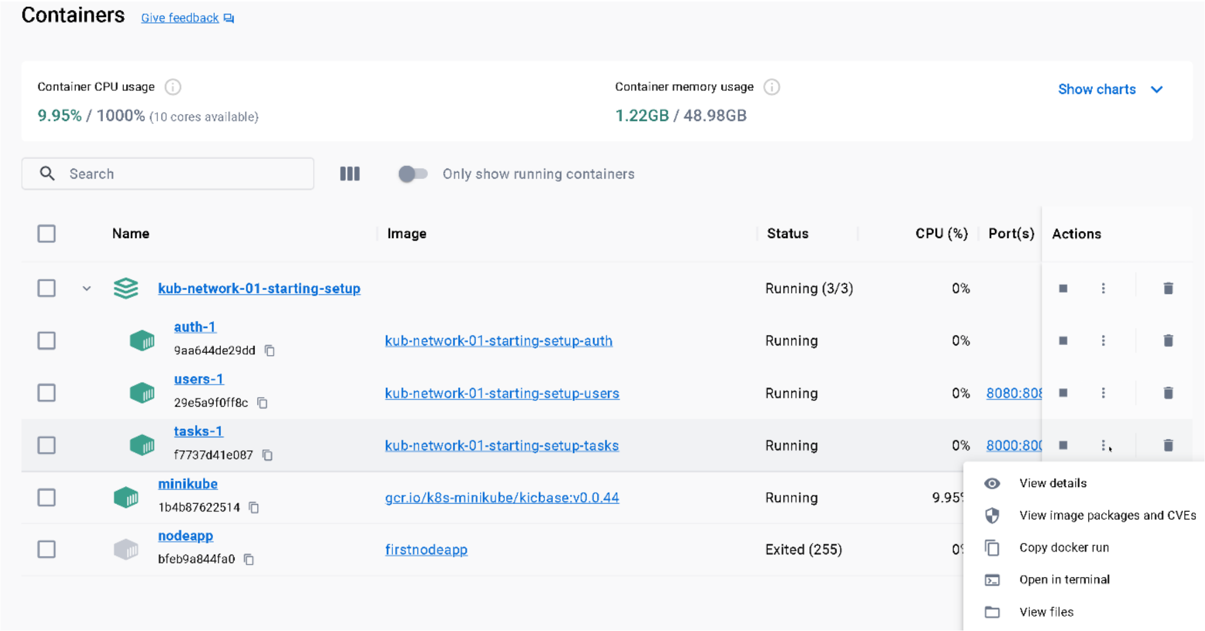
Body: JSON

```
{
    "message": "Tasks loaded.",
    "tasks": [
        {
            "title": "Fly in the Sky",
            "text": "My First Task"
        }
    ]
}
```

You can continue to send POST requests with further random data for `title` and `text` properties in JSON content to this API endpoint and subsequently retrieve the list of tasks so far with a GET request to the same endpoint.

If you inspect `tasks-app.js`, you will see that the data in the `title` and `text` properties of the JSON content send the POST request to the `tasks` app is being stored in the file `tasks.txt` in the folder specified by the environment variable TASKS_FOLDER. This variable in turn currently has the value `tasks` as specified in the main Docker Compose file in the top level directory.

If you have Docker Desktop installed, you should be able to see the service for the `tasks` app visible in the Containers section of the main dashboard UI, nested under the top level Compose project. Clicking on the ellipsis menu gives you list of options, including viewing the file system as well as opening a terminal in the container.

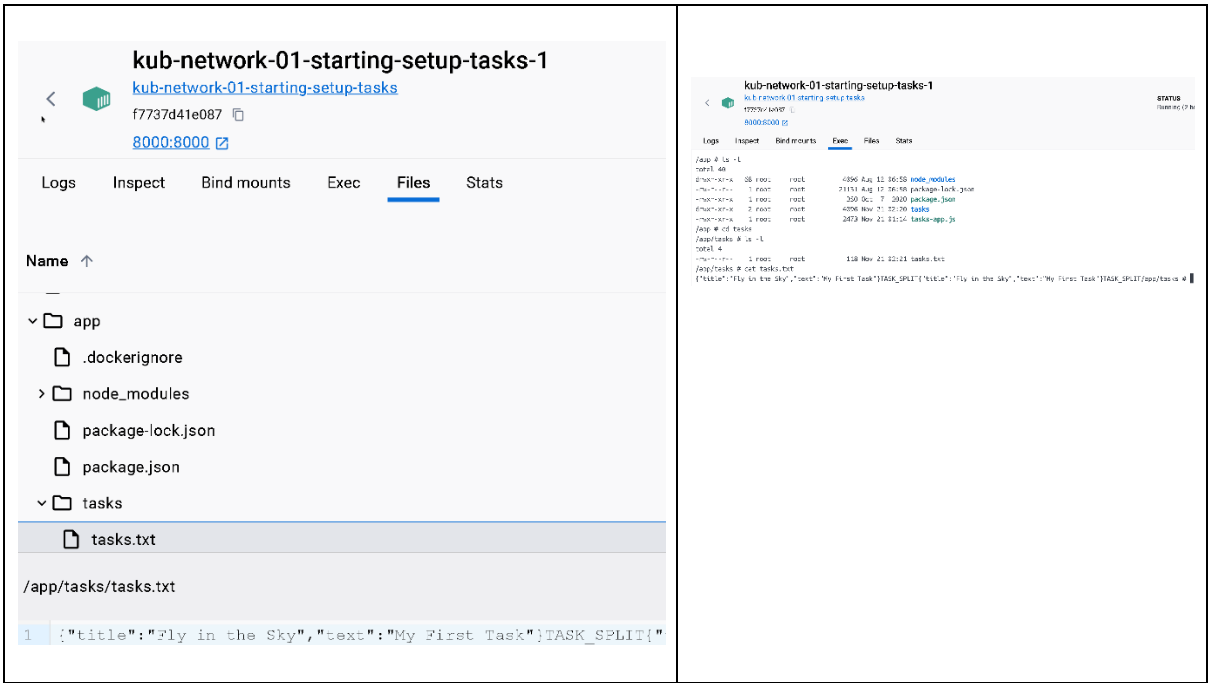


Container CPU usage 9.95% / 1000% (10 cores available)

Container memory usage 1.22GB / 48.98GB

□	Name	Image	Status	CPU (%)	Port(s)
□	kub-network-01-starting-setup		Running (3/3)	0%	
□	auth-1 9aa644de29dd	kub-network-01-starting-setup-auth	Running	0%	
□	users-1 29e5a9f0ff8c	kub-network-01-starting-setup-users	Running	0%	8080:808
□	tasks-1 f7737d41e087	kub-network-01-starting-setup-tasks	Running	0%	8000:800
□	minikube 1b4b87622514	gcr.io/k8s-minikube/kicbase:v0.0.44	Running	9.95	
□	nodeapp bfeb9a844fa0	firstnodeapp	Exited (255)	0	

You should be able to view the file system of the `tasks` app container to verify that it contains the directory `/app/tasks` with the file `tasks.txt` containing the JSON content separated by a delimiter marker TASK_SPLIT (which was placed there in by the tasks app).
You can also check this from the CLI using the standard Linux command to switch directories (`cd`), view directory content (`ls -l`) and view file content (`cat tasks.txt`)

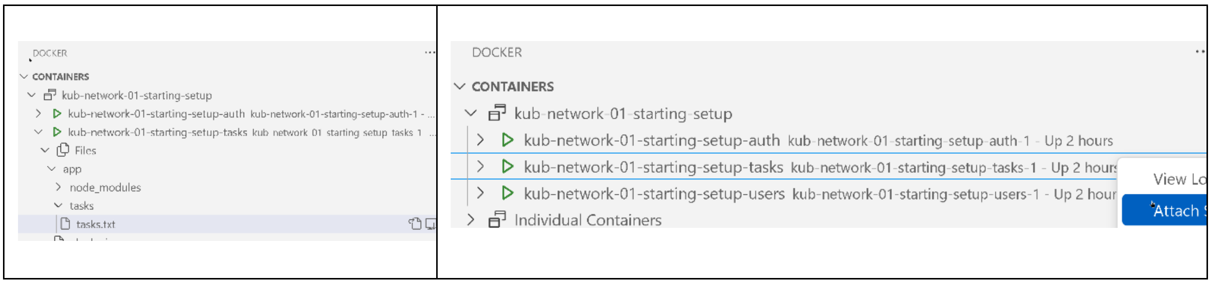


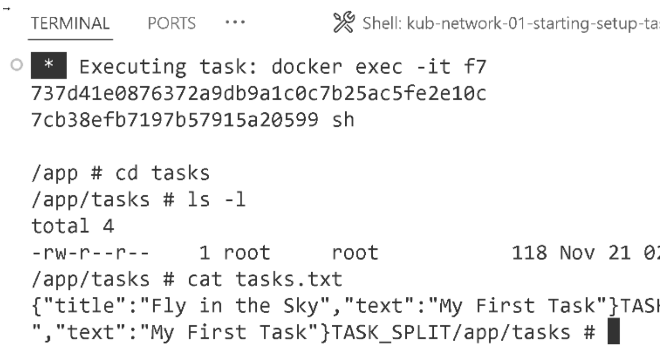
/app/tasks/tasks.txt

```
{"title":"Fly in the Sky","text":"My First Task"}TASK_SPLIT{"
```

```
/app # ls -l
total 40
drwxr-xr-x   58 root     root          4096 Aug 12 06:58 node_modules
-rw-r--r--    1 root     root         21151 Aug 12 06:58 package-lock.json
-rwxr-xr-x    1 root     root           350 Oct  7  2020 package.json
drwxr-xr-x    2 root     root          4096 Nov 21 02:20 tasks
-rwxr-xr-x    1 root     root          2473 Nov 21 01:14 tasks-app.js
/app # cd tasks
/app/tasks # ls -l
total 4
-rw-r--r--    1 root     root           118 Nov 21 02:21 tasks.txt
/app/tasks # cat tasks.txt
{"title":"Fly in the Sky","text":"My First Task"}TASK_SPLIT{"title":"Fly in the Sky","text":"My First Task"}TASK_SPLIT/app/tasks #
```

If you have the Docker extension for VS Code installed, the functionality of viewing the file system of the container / service and opening a terminal in it is also available.


DOCKER
CONTAINERS
kub-network-01-starting-setup
kub-network-01-starting-setup-auth kub-network-01-starting-setup-auth-1 - ...
kub-network-01-starting-setup-tasks kub network 01 starting setup tasks 1 ...
Files
app
node_modules
tasks
tasks.txt
DOCKER
CONTAINERS
kub-network-01-starting-setup
kub-network-01-starting-setup-auth kub-network-01-starting-setup-auth-1 - Up 2 hours
kub-network-01-starting-setup-tasks kub-network-01-starting-setup-tasks-1 - Up 2 hours
kub-network-01-starting-setup-users kub-network-01-starting-setup-users-1 - Up 2 hour
Individual Containers
View Lo
Attach S



```
 *  Executing task: docker exec -it f7
737d41e0876372a9db9a1c0c7b25ac5fe2e10c
7cb38efb7197b57915a20599 sh

/app # cd tasks
/app/tasks # ls -l
total 4
-rw-r--r--    1 root     root           118 Nov 21 0
/app/tasks # cat tasks.txt
{"title":"Fly in the Sky","text":"My First Task"}TAS
","text":"My First Task"}TASK_SPLIT/app/tasks #
```

If everything is working, you can bring down all the services in this Compose project with:

```
docker compose down
```

4.2 Creating a deployment

We will start migrating the different containers comprising this app to Kubernetes step by step.

We will start first with migrating `users` app to a pod in Kubernetes. We will make some changes to `users-app.js` initially to comment out the code that is making HTTP calls to the `auth-api` app running at port 80 and instead substitute some dummy values which represent the response that would normally be obtained from those calls. The reason we do is because when we start up this app in a container in a pod, it will be the only container in that pod and any calls to the `auth-api` app will cause the app to crash.

`users-app.js`

```
….
….

  try {
    //const hashedPW = await axios.get('http://auth/hashed-password/' +
password);
    // Comment out the previous request
    const hashedPW = "dummy text";
    // since it's a dummy service, we don't really care for the hashed-pw
either
    console.log(hashedPW, email);
    res.status(201).json({ message: 'User created!' });
  } catch (err) {

…..
….

  // normally, we'd find a user by email and grab his/ her ID and hashed
password
  const hashedPassword = password + '_hash';

  // const response = await axios.get(
  //   'http://auth/token/' + hashedPassword + '/' + password
  // );

  // Replace with a dummy response with status 200 and a dummy token

  const response = {status: 200, data : { token : 'abc'} };
```

Navigate into the `users-api`, and use the Dockerfile there to build a new image to incorporate these code changes with:

```
docker build -t dockerhubaccount/kub-demo-users .
```

Check that the image was successfully built through a UI or through the command:

```
docker images
```

Push this newly built image to your DockerHub account:

```
docker push dockerhubaccount/kub-demo-users
```

In the project root folder: `kub-network-01-starting-setup`
create a subfolder `kubernetes` to store all your YAML configuration manifest files (make sure it is in the project root folder)

In that folder, create a deployment file for this app:

`users-deployment.yaml`

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: users-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: users
  template:
    metadata:
      labels:
        app: users
    spec:
      containers:
        - name: users
          image: dockerhubaccount/kub-demo-users
```

Navigate into this directory and use this YAML to start the deployment with:

```
kubectl apply -f users-deployment.yaml
```

Check that it is up and running with:

```
kubectl get deployments
```

```
kubectl get pods
```

We need to now create a service to allow us to expose the `users-api` app in the pod in the deployment that we have just created for external access. We will place this in the same `kubernetes` folder that we created earlier:

`users-service.yaml`

```
apiVersion: v1
kind: Service
metadata:
  name: users-service
spec:
  selector:
    app: users
  type: LoadBalancer
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 8080
```

The key fields here are:

- `port: 8080` - specifies the port on which the service is exposed to external clients. Clients that wish to send traffic to the service will direct their HTTP requests to this port.
- `targetPort: 8080` - specifies the internal port on the pod that the service will forward traffic to. In this case, `users-app.js` in the single container within the pod listens on this port, and so will receive all incoming requests that are directed to this service.

Then create the service with:

```
kubectl apply -f users-service.yaml
```

Check that it is started:

```
kubectl get services
```

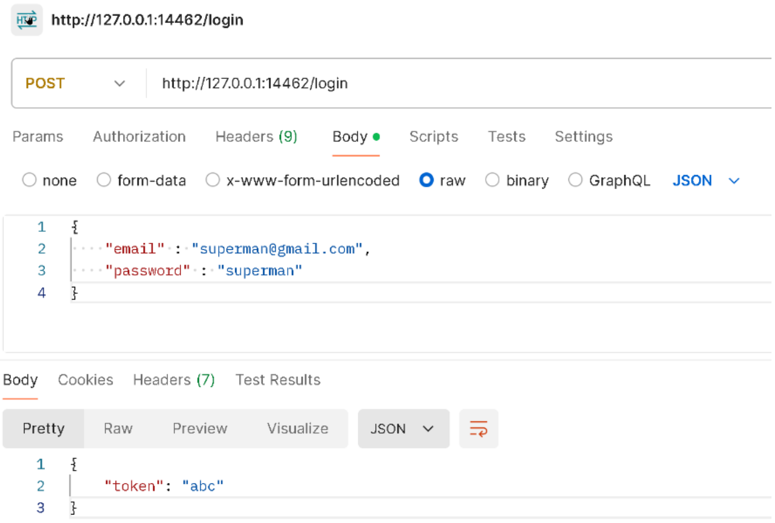
Then to allow external access to this service, we use Minikube with:

```
minikube service users-service
```

As we have already seen, Minikube provides a specific way to access the port on the NodePort: it creates a tunnel from a port on localhost to the NodePort. This localhost port is randomly assigned from the range of free ports on the local machine. You can then use the loopback address (`127.0.0.1`) or `localhost` at this tunnelling port to forward HTTP traffic to `users-deployment`, which automatically is accessed through a browser tab when you use the `minikube service` command.

Use this URL and send a POST request with Postman to this app at the API endpoint `/login` with a JSON content for a dummy username/password pair and verify that you can get back the dummy token that you hardcoded into `users-app.js` earlier, for e.g.

```
{
    "email" : "superman@gmail.com",
    "password" : "superman"
}
```



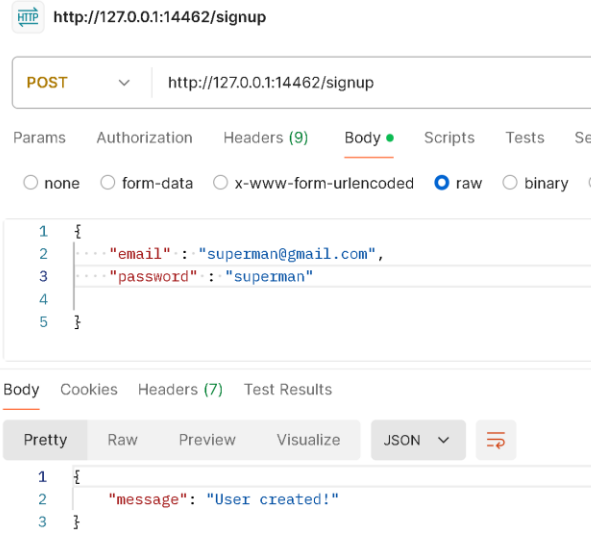
http://127.0.0.1:14462/login

POST http://127.0.0.1:14462/login

```
{
    "email" : "superman@gmail.com",
    "password" : "superman"
}
```

```
{
    "token": "abc"
}
```

We should also get back the dummy successful response that we hardcoded earlier in our modifications to `users-app.js` if we send a POST request with similar JSON content for a username/password pair to the API endpoint: `/signup`



http://127.0.0.1:14462/signup

POST http://127.0.0.1:14462/signup

```
{
    "email" : "superman@gmail.com",
    "password" : "superman"

}
```

```
{
    "message": "User created!"
}
```

This will verify that the app in the container in the single pod of that deployment is now reachable externally via the service that we had just created.

5 Intra-pod container communication

We will now run the auth app in a container in the same pod as the users app container in order to demonstrate communication between multiple pods in the same container.

For pod-internal communication, Kubernetes uses the localhost value to reference the internal network system of the pod. Any app wishing to send outgoing HTTP requests (for e.g. the users app) will specify this value (localhost) as well as a specific port in the initial portion of the URL in all its outgoing HTTP requests, for e.g.

http://localhost:9090/api/some-end-point

Any other container that has an app listening on this port 9090 (for e.g. the auth app) will then be able to receive and process these incoming HTTP requests on the matching path portion of the API endpoint.

Notice that this approach is different from Docker Compose, where we would use the name of the service (for e.g. auth) that we wish to target in the outgoing HTTP request, rather than localhost.

As we prepare to run the auth app in a container in the same pod as the users app, we now need to revert our code in users-app.js back to its original form as well as provide the ability to access the contact address for the auth app (i.e. localhost) via an environment variable:

users-app.js

```
….
….

  try {

  try {

    // Specify name of auth-api app via environment variable

    const hashedPWAPI = `http://${process.env.AUTH_ADDRESS}/hashed-
password/` + password;

    console.log("Sending GET request to ", hashedPWAPI)
    const response = await axios.get(hashedPWAPI);

    // const hashedPW = "dummy text";
    // since it's a dummy service, we don't really care for the hashed-pw
either
    console.log("Received back a hashedPW : " + response.data.hashedPassword
+ " corresponding to email " + email);

    res.status(201).json({ message: 'User created!' });
  } catch (err) {
```

```
….
….
….
….

  // normally, we'd find a user by email and grab his/ her ID and hashed
password
  // here we just create a simple hardcoded hashed password
  const hashedPassword = password + '_hash';

  // Specify name of auth-api app via environment variable

  const tokenAPI = `http://${process.env.AUTH_ADDRESS}/token/` +
hashedPassword + '/' + password;

  console.log("Sending GET request to ", tokenAPI);

  const response = await axios.get( tokenAPI );
  // Replace with a dummy response with status 200 and a dummy token
  // const response = {status: 200, data : { token : 'abc'} };

  if (response.status === 200) {
    console.log ("Received back token : " + response.data.token);
    return res.status(200).json({ token: response.data.token });
  }

….
….
….
….
```

We now have to rebuild a new image for the `users` app that incorporates these latest changes and make it available on our DockerHub account, since Kubernetes deployment YAML configuration will by default retrieve images to build containers in pods from DockerHub.

Open a new Powershell terminal (keeping it separate for this purpose) inside the `users-api` folder.

Use the Dockerfile there to rebuild the image to incorporate these latest changes:

```
docker build -t dockerhubaccount/kub-demo-users .
```

After a successful build, push this newly built image to your DockerHub account:

```
docker push dockerhubaccount/kub-demo-users
```

We can then modify the top level `docker-compose.yaml` to specify the environment variable that we used in `users-app.js`, which will allow us to substitute a different value if we were to ever start the users app again as a container / service as a component of a Compose project in the future. Here AUTH_ADDRESS references the name of the service we wish to contact (i.e. the auth service).

```
..
…

  users:
    build: ./users-api
    environment:
      AUTH_ADDRESS: auth
    ports:
      - "8080:8080"
  tasks:

…
…
```

Similarly, as we are now going to incorporate the auth app as a new container alongside the users app in our pod, we also need to build an image for it and make this available on our DockerHub account, since Kubernetes deployment YAML configuration will by default retrieve images to build containers in pods from DockerHub.

Open a new Powershell terminal (keeping it separate for this purpose) inside the auth-api folder:

Build the Dockerfile in there and generate an image to push to DockerHub:

```
docker build -t dockerhubaccount/kub-demo-auth .
```

When the new image is rebuilt, we can then push it to DockerHub with:

```
docker push dockerhubaccount/kub-demo-auth
```

We now modify the existing users-deployment.yaml in Kubernetes with the following modifications:

- specify a container for the auth app that we wish to now add into the single pod containing the users app.
- specify the name of the environment variable AUTH_ADDRESS which will be passed to our users-app.js, where our previously modified code will access it. This name will be localhost, which is the default name that Kubernetes uses for the internal networking system of the pod to allow an app in a container to communicate with another app in a different container in the same pod

users-deployment.yaml

```
…
…
…
```

```
template:
  metadata:
    labels:
      app: users
  spec:
    containers:
      - name: users
        image: dockerhubaccount/kub-demo-users
        ports:
          - containerPort: 8080
        env:
          - name: AUTH_ADDRESS
            value: localhost
      - name: auth
        image: dockerhubaccount/kub-demo-auth
        ports:
          - containerPort: 80
```

Some things to note:

Although we specify the `containerPort` value here, this is entirely optional and primarily for documentation purposes since the apps will be listening on the specific container port specified explicitly in their code (`app.listen(xxx);`)

Note that the current service `users-service` redirects incoming traffic to it to the internal port of the Pod at 8080 (as provided by `targetPort` in `users-service.yaml`), which is currently being listened to by `users` app. This means that the `auth` app is not able to receive traffic from an external client (such as Postman) that is outside the pod.

We can expose the `auth` app by creating another service that redirects traffic to its container port of 80 (similar to `users-service.yaml`), however we will not do that here since the intended design for this system is for this app is to be contactable only by the `users` app within the pod, and not by an external application. Services are only necessary to expose pods within a deployment whose containers will be communicated with by an external client or entity outside that pod (such as our browser or Postman client).

In a separate new Powershell terminal, navigate back to the `kubernetes` folder to apply these configuration changes to the deployment:

```
kubectl apply -f users-deployment.yaml
```

Check that the new pod is up and running, and the previous one has being terminated:

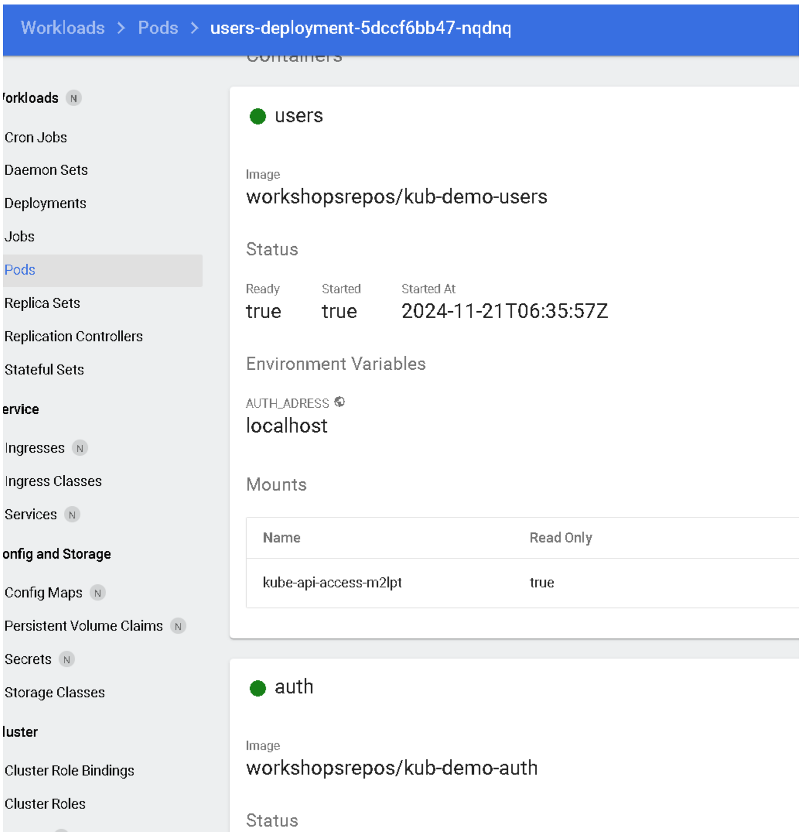
```
kubectl get pods
```

To get more info on the single pod that is running with 2 containers (`auth` and `users`):

```
kubectl describe pod users-deployment
```

You should be able to see info on both containers.
These includes the image they were created from, the events involved in pulling these images from their DockerHub locations and building them, and environment variables that are associated with them (`AUTH_ADDRESS: localhost` for the case of `users`)

You can also view this info from the Minikube dashboard as well.



● users

Image
workshopsrepos/kub-demo-users

Status

Ready	Started	Started At
true	true	2024-11-21T06:35:57Z

Environment Variables

AUTH_ADRESS
localhost

Mounts

Name	Read Only
kube-api-access-m2lpt	true

● auth

Image
workshopsrepos/kub-demo-auth

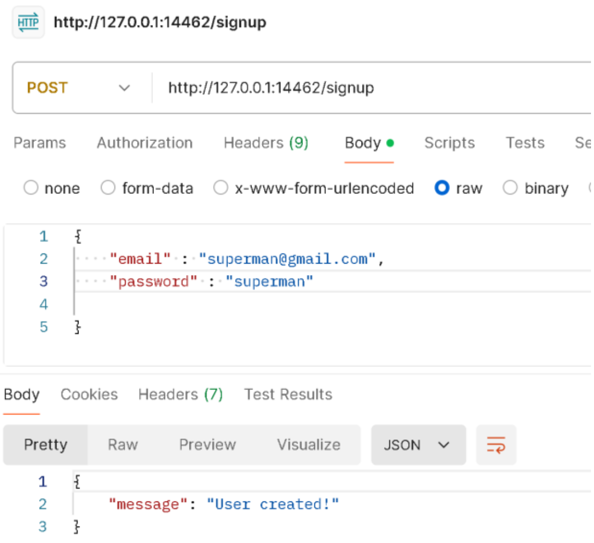
Status

Now we repeat the actions from a previous lab session:

Using the URL and port number exposed by the service for the `users-deployment` pod, send a POST request to the `user-api` app at the API endpoint `/signup` with a JSON content for a username/password pair

```
{
"email" : "superman@gmail.com",
"password" : "superman"
}
```

Verify that you can get back a successful message.
This indicates that the `user-api` app was able to successfully communicate with the `auth-api` app that are both running in two different containers within the same pod.



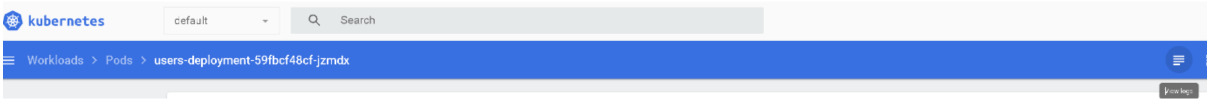
POST http://127.0.0.1:14462/signup

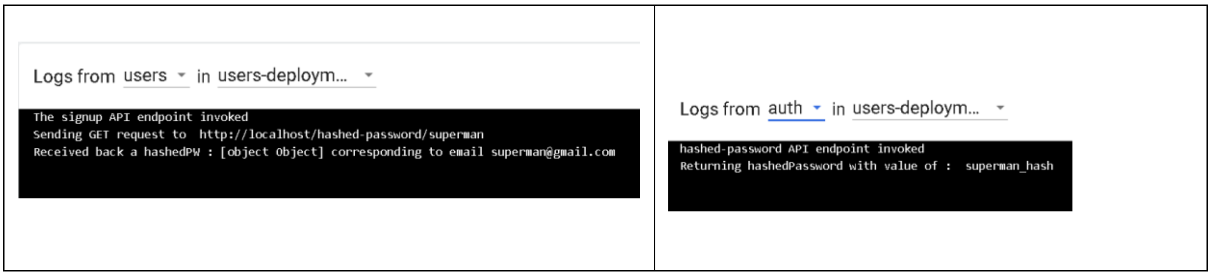
```
{
    "email" : "superman@gmail.com",
    "password" : "superman"

}
```

```
{
    "message": "User created!"
}
```

You can verify the actual API endpoints that were called in both apps (`users` and `auth`) which is displayed in `console.log` statements and are available by examining the log output from these two containers in the pods.

The log output for both containers is accessible in the Minikube dashboard section for single Pod that contains them via an option at the upper right hand corner.

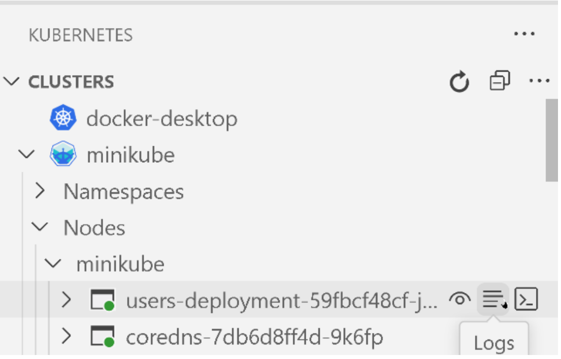


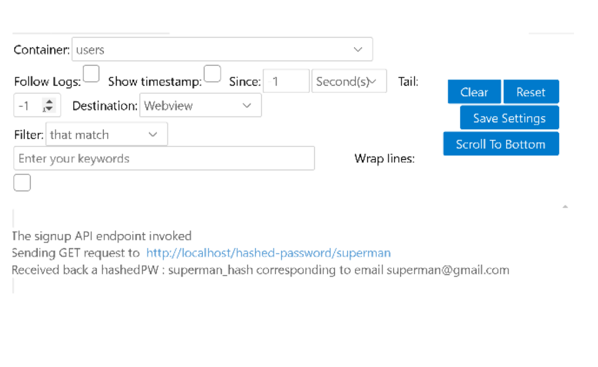

Logs from users in users-deploym...
Logs from auth in users-deploym...

```
The signup API endpoint invoked
Sending GET request to  http://localhost/hashed-password/superman
Received back a hashedPW : [object Object] corresponding to email superman@gmail.com
```

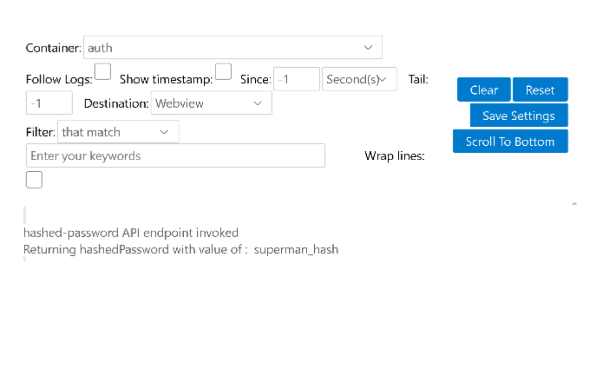
```
hashed-password API endpoint invoked
Returning hashedPassword with value of :  superman_hash
```

If you have the Kubernetes extension for VS code installed, this also provide a logs option for you to examine the log output from all of the existing containers running within that pod.





```
The signup API endpoint invoked
Sending GET request to  http://localhost/hashed-password/superman
Received back a hashedPW : superman_hash corresponding to email superman@gmail.com
```



```
hashed-password API endpoint invoked
Returning hashedPassword with value of :  superman_hash
```

Finally, as a last option, you can also examine the logs from any particular container running in a given pod with the CLI command:

```
kubectl logs pod-name -c container-name
```

If using this command, ensure that you have your *pod-name* and *container-name* specified correctly based on the info from the `kubectl describe pod users-deployment` command.

The key point to note in the log output is the URL used in the GET request sent from the users app:

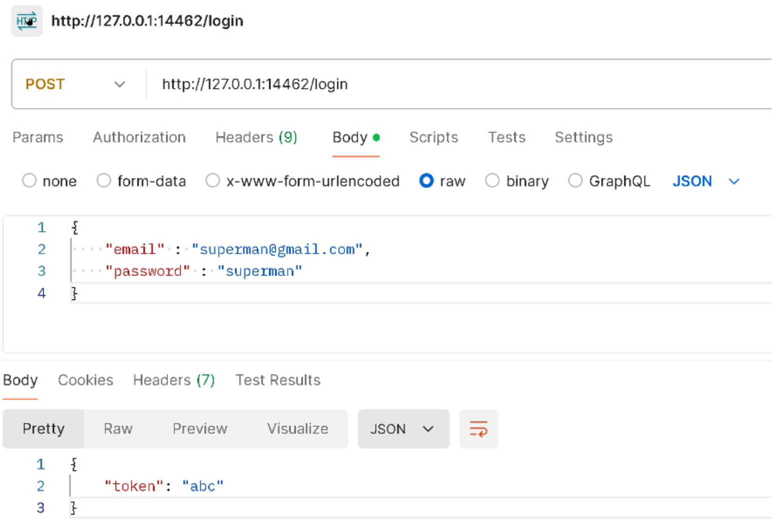
```
Sending GET request to  http://localhost/token/superman_hash/superman
```

Notice here that we do not explicitly specify the port 80 that the `auth` app is listening on in the URL. This is because the default port for any domain name (including localhost) is port 80, which is the default port for HTTP traffic for a web server and is implicitly included if not port is specified, such as is the case here. If we had however bound the `auth` app to another port such as 9090 for e.g. (`app.listen(9090);`), then the GET request would need to explicitly include that port as well.

As a general rule, putting appropriate `console.log` statements in your app code in order to examine their output when they are actually running in containers within a pod is very useful in helping to trace program logic and debugging any unexpected app behaviour.

Verify that we will also get back a token if we send a POST request with identical JSON content for a username/password pair to the same URL of the service for the `users-deployment` pod, but at a different API endpoint: `/login`

```
{
"email" : "superman@gmail.com",
"password" : "superman"
}
```



http://127.0.0.1:14462/login

POST http://127.0.0.1:14462/login

Body: ● raw JSON

```
{
    "email" : "superman@gmail.com",
    "password" : "superman"
}
```

Pretty JSON

```
{
    "token": "abc"
}
```

Again, you can verify the actual API endpoints that were called in both apps (`users` and `auth`) which is displayed in `console.log` statements and are available by examining the log output from these two containers in the pods: either through Minikube Dashboard UI or the VS Code Kubernetes extension or directly through a CLI command as explained previously.

Both the successful responses indicating that the code within the `user` app that executes the logic for these two API endpoints (`/signup` and `/login`) has successfully executed HTTP REST API calls to the two API endpoint (`/token` and `/hashed-password`) in the `auth` app running in a different container in the same pod. This communication was achieved using the address the address `localhost` that was passed by the environment variable `AUTH_ADDRESS` specified in `users-deployment.yaml` and subsequently accessed by the code in `users-app.js`

6 Creating deployments for multiple pods

In the previous scenario, we have utilized the multiple container – single pod model. This is not a very common model, but it can be used under the following situation:

Containers have apps that are tightly coupled, for e.g. one app is the main backend service and the other app performs closely related functionality such as logging, monitoring, caching data or providing simple authentication (such as is the case here). These apps are likely to share data, which can be easily accomplished by sharing storage volumes. Having multiple containers within a single app also reduces the overhead of inter-pod communication.

Drawbacks however include:

- Complexity: Managing lifecycle and resource allocation is harder as containers within a pod share resources.
- Scaling Limitations: All containers in a pod scale together, even if only one container's resource needs increase.

The more common model is a single container within a single pod, because of the following advantages that it offers:

- Simplicity: Single-container pods are simpler to manage, monitor, and troubleshoot.
- Isolation: Each pod focuses on a single responsibility, adhering to the microservices architecture principle.
- Scaling: Scaling is straightforward since each pod corresponds to a single service/component.

Here we will create separate pods and a corresponding deployment to place the `users` and `auth` apps, so that we have the single container – single pod model.

Stop the currently running deployment:

```
kubectl delete deployment users-deployment
```

Within the `kubernetes` directory, create a new `auth-deployment.yaml` file specifically to deploy the pod containing the `auth` app:

`auth-deployment.yaml`

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: auth-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: auth
  template:
    metadata:
      labels:
        app: auth
    spec:
      containers:
        - name: auth
          image: dockerhubaccount/kub-demo-auth
```

We also refactor the existing `users-deployment.yaml` to remove the creation of the container containing the `auth-api` app from there, since we now have a separate deployment file for it.

`users-deployment.yaml`

```
…
```

```
…

    spec:
      containers:
        - name: users
          image: dockerhubaccount/kub-demo-users
          ports:
            - containerPort: 8080
          env:
            - name: AUTH_ADDRESS
              value: localhost
```

Next create a configuration manifest for a service to expose the `auth-api` pod internally.

`auth-service.yaml`

```
apiVersion: v1
kind: Service
metadata:
  name: auth-service
spec:
  selector:
    app: auth
  type: ClusterIP
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
```

The key point is the use of the ClusterIP service type. ClusterIP services are only accessible within the cluster. It assigns a virtual IP that can be used by other pods or services within the cluster to communicate with it. This is the default service type if none is specified. The LoadBalancer and Nodeport services can be accessed by an external client and therefore can redirect external traffic (such as from the web browser or the Postman client) to a pod.

7 Interpod communication

Once we start the deployments and the corresponding services for the `auth` and `users` app that will now be running in 2 separate pods, we will need to ensure that the apps in these 2 pods can still communicate with each other.

The most straight forward approach is for the application code in the pods to use the internal IP address of the pods when communicating. Each pod is allocated an internal IP address when it is created that lasts for its lifetime; this can be verified via:

```
kubectl get pods -o wide
```

However, it is not common practice to use the internal IP address. Instead, we use services to intermediate communication between pods for the following reasons:

a) Pod IPs are ephemeral: Pods can be rescheduled or restarted, resulting in a change in their IP address. IP addresses of services remain stable, even while the pods that they expose are rescheduled or restarted. Thus, they provide a stable endpoint to connect to, abstracting away the changing Pod IPs.
b) Scalability: Services can automatically load-balance traffic between multiple Pods in a deployment or replica set. This functionality will need to be manually implemented if Pod-to-Pod communication is taking place.

7.1 Using internal IP address of the service

There are several ways to accomplish this. One is to utilize the internal IP address associated with that service directly in application code. Services typically have a stable internal IP address which remains fixed for the duration of the service lifetime (i.e up until the time when the service is deleted and recreated). This is definitely true for ClusterIP and is also true for NodePort and LoadBalancer service types, although the external IP address for these 2 services may change depending on the external deployment environment (for e.g. dynamic cloud setups).

One way to determine the fixed IP address for a particular service is to generate the service itself.
We could do this for the auth deployment and its associated service with:

```
kubectl apply -f auth-service.yaml -f auth-deployment.yaml
```

Check for the deployment, pod and services with:

```
kubectl get deployments

kubectl get pods

kubectl get services
```

The listing of the services should enable us to obtain the internal IP address (CLUSTER-IP) that the service is accessible on within the Kubernetes cluster.

```
NAME            TYPE           CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
auth-service    ClusterIP      10.100.205.239   <none>        80/TCP           34s
kubernetes      ClusterIP      10.96.0.1        <none>        443/TCP          27h
users-service   LoadBalancer   10.103.74.41     <pending>     8080:30735/TCP   5h
```

Then we can use this IP address for the AUTH_ADDRESS environment variable in users-deployment.yaml

users-deployment.yaml

```
…..
…..
……

    spec:
      containers:
        - name: users
          image: dockerhubaccount/kub-demo-users:latest
          env:
            - name: AUTH_ADDRESS
              value: "10.100.205.239"
```

Now we can apply this new configuration changes to recreate the pod with the `users` app:

```
kubectl apply -f users-deployment.yaml
```

If there are any issues with the associated service `users-service`, you can recreate it again with:

```
kubectl apply -f users-service.yaml
```

Now if you check the deployments and pods:

```
kubectl get deployments
```

```
kubectl get pods
```

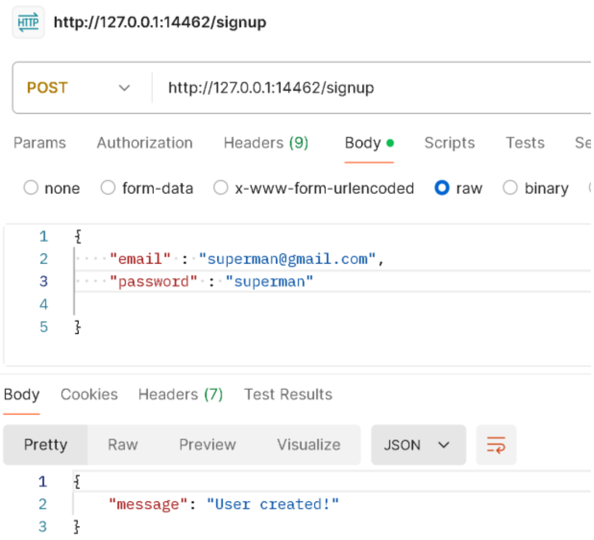
You should two separate deployments each, with each deployment having exactly one pod (for the `auth` app and `users` app)

Finally, to test that everything is still working, we repeat the previous actions from a previous lab session:

Using the URL and port number exposed by the `users-service` for the `users-deployment` pod, send a POST request to the `user-api` app at the API endpoint `/signup` with a JSON content for a username/password pair

```
{
"email" : "superman@gmail.com",
"password" : "superman"
}
```

Verify that you can get back a successful message.
This indicates that the `user-api` app was able to successfully communicate with the `auth-api` app via the IP address for the service that exposes the `auth-api` app. Both apps are now running in two different pods within the same node, each with their own internal IP address.



http://127.0.0.1:14462/signup

POST http://127.0.0.1:14462/signup

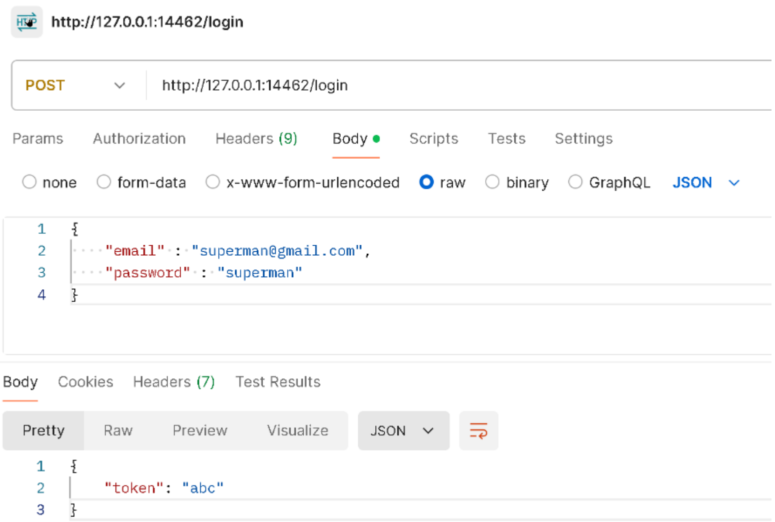
```
{
    "email" : "superman@gmail.com",
    "password" : "superman"

}
```

```
{
    "message": "User created!"
}
```

Just as before, you can verify the actual API endpoints that were called in both apps (`users` and `auth`) which is displayed in `console.log` statements and are available by examining the log output from these two pods: either through Minikube Dashboard UI or the VS Code Kubernetes extension or directly through a CLI command as explained previously. In particular, you should see the GET request from the `users` app targeting the IP Address that you specified for the `AUTH_ADDRESS` environment variable in `users-deployment.yaml`

```
Sending GET request to  http://10.100.205.239/hashed-password/superman
```

Verify that we will also get back a token if we send a POST request with similar JSON content for a username/password pair to the API endpoint: `/login`



http://127.0.0.1:14462/login

POST http://127.0.0.1:14462/login

```
{
    "email" : "superman@gmail.com",
    "password" : "superman"
}
```

```
{
    "token": "abc"
}
```

Just as before, you can verify the actual API endpoints that were called in both apps (`users` and `auth`) which is displayed in `console.log` statements and are available by examining the log output from these two pods: either through Minikube Dashboard UI or the VS Code Kubernetes extension or directly through a CLI command as explained previously.

Both the successful responses indicating that the code within the user app that handles these two API endpoints (/signup and /login) has successfully executed HTTP API calls to the auth running in a different pod on the same node, using the IP address for the service that exposes the auth app (auth-service) with this IP address being passed to users-app.js via the environment variable AUTH_ADDRESS specified in users-deployment.yaml

7.2 Using automatically generated environment variables

Kubernetes provides automatically generated environment variables in applications running inside pods with information about all services running in the cluster. For instance, with the auth-service and users-service running, Kubernetes automatically generates environment variables containing service information like their IP addresses.

In a Node.js application, you can access these variables using process.env. The variable names follow a specific pattern: the service name in all caps, with dashes replaced by underscores, appended with _SERVICE_HOST. For example, for auth-service, the variable name would be AUTH_SERVICE_SERVICE_HOST. This convention applies to all services.

We can now make a change in the code implementation of users-app.js to access this auto-generated environment variable, whose front portion is based on the name specified in auth-service.yaml (name: auth-service)

users-app.js

```
….
…
…

  // normally, we'd find a user by email and grab his/ her ID and hashed
password
  // here we just create a simple hardcoded hashed password
  const hashedPassword = password + '_hash';

  // Specify name of auth-api app via Kubernetes
  // auto generated environment variable xxxx_SERVICE_HOST

  const tokenAPI = `http://${process.env.AUTH_SERVICE_SERVICE_HOST}/token/`
+ hashedPassword + '/' + password;

  console.log("Sending GET request to ", tokenAPI);

….
…
…
```

With this change, our code implementation now uses two different ways to identify the auth pod in the two separate HTTP REST API calls:

```
….
…
…
    // Specify name of auth-api app via environment variable

    const hashedPWAPI = `http://${process.env.AUTH_ADDRESS}/hashed-
password/` + password;

    console.log("Sending GET request to ", hashedPWAPI)
    const response = await axios.get(hashedPWAPI);

….
…
…

  // Specify name of auth-api app via Kubernetes
  // auto generated environment variable xxxx_SERVICE_HOST

  const tokenAPI = `http://${process.env.AUTH_SERVICE_SERVICE_HOST}/token/`
+ hashedPassword + '/' + password;

  console.log("Sending GET request to ", tokenAPI);

  const response = await axios.get( tokenAPI );
```

Return back to the `users-api` folder and rebuild the image to incorporate the latest code changes with:

```
docker build -t dockerhubaccount/kub-demo-users .
```

Once it is built, push it to your DockerHub account:

```
docker push dockerhubaccount/kub-demo-users
```

Return back to the `kubernetes` folder, and delete the deployment associated with `users-deployment.yaml` and generate it again with:

```
kubectl delete deployment users-deployment

kubectl apply -f users-deployment.yaml
```

Check that the associated deployment and pod are eventually up and running:

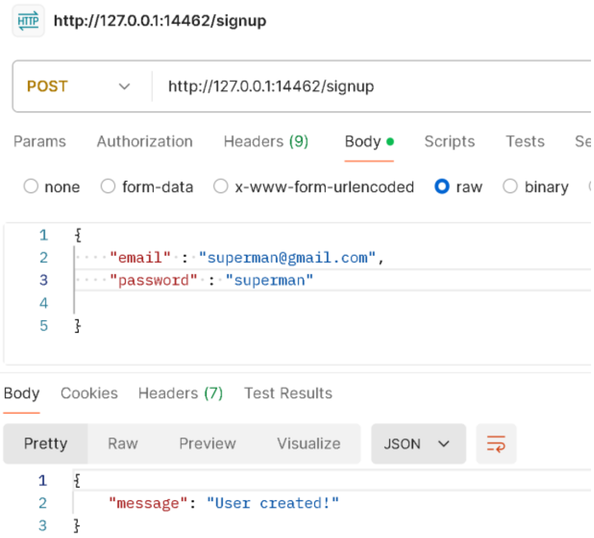
```
kubectl get deployments

kubectl get pods
```

Again, to test that everything is still working, we repeat the previous actions from a previous lab session:

Using the URL and port number exposed by the `users-service` for the `users-deployment` pod, send a POST request to the `user-api` app at the API endpoint `/signup` with a JSON content for a username/password pair

```
{
"email" : "superman@gmail.com",
"password" : "superman"
}
```

Verify that you can get back a successful message.
This indicates that the `user-api` app was able to successfully communicate with the `auth-api` app via the IP address for the service that exposes the `auth-api` app. Both apps are now running in two different pods within the same node, each with their own internal IP address.



http://127.0.0.1:14462/signup

POST http://127.0.0.1:14462/signup

○ none ○ form-data ○ x-www-form-urlencoded ● raw ○ binary

```
{
    "email" : "superman@gmail.com",
    "password" : "superman"

}
```

JSON

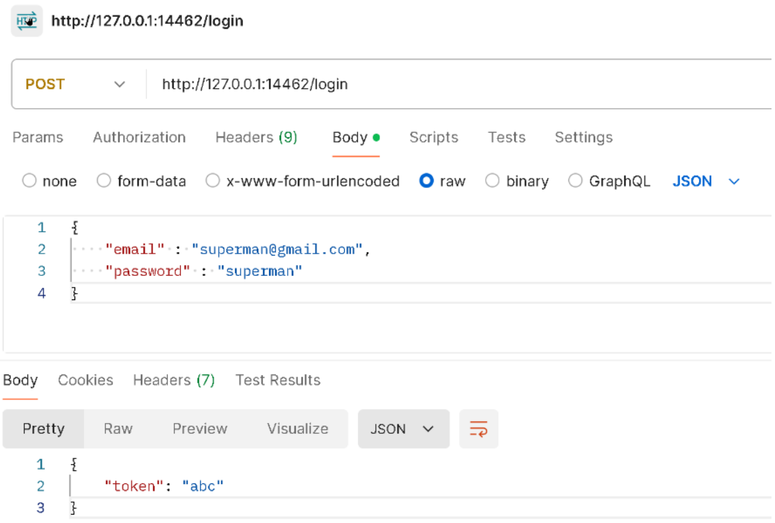
```
{
    "message": "User created!"
}
```

Just as before, you can verify the actual API endpoints that were called in both apps (`users` and `auth`) which is displayed in `console.log` statements and are available by examining the log output from these two pods: either through Minikube Dashboard UI or the VS Code Kubernetes extension or directly through a CLI command as explained previously.

In particular, you should see the GET request from the `users` app targeting the IP address associated with the auto-generated AUTH_SERVICE_SERVICE_HOST environment variable that you placed in `user-app.js` earlier. This is exactly identical to the hardcoded IP Address that you specified for the `AUTH_ADDRESS` environment variable in `users-deployment.yaml`

```
Sending GET request to  http://10.100.205.239/hashed-password/superman
```

Verify as well that we will also get back a token if we send a POST request with similar JSON content for a username/password pair to the API endpoint: `/login`



POST http://127.0.0.1:14462/login

```
{
    "email" : "superman@gmail.com",
    "password" : "superman"
}
```

```
{
    "token": "abc"
}
```

Just as before, you can verify the actual API endpoints that were called in both apps (`users` and `auth`) which is displayed in `console.log` statements and are available by examining the log output from these two pods: either through Minikube Dashboard UI or the VS Code Kubernetes extension or directly through a CLI command as explained previously.

In particular, you should see the GET request from the `users` app targeting the IP address associated with the auto-generated AUTH_SERVICE_SERVICE_HOST environment variable that you placed in `user-app.js` earlier. This is exactly identical to the hardcoded IP Address that you specified for the `AUTH_ADDRESS` environment variable in `users-deployment.yaml`

```
Sending GET request to  http://10.97.249.204/token/superman_hash/superman
```

Both the successful responses indicating that the two approaches of specifying the IP address for the `auth-service` are working for the purposes of interpod communication.

One important thing to note is that this auto-generated AUTH_SERVICE_SERVICE_HOST environment variable is only meaningful within a Kubernetes environment. If you wish to retain this environment variable in your source code for `users-app.js` but still be able to deploy this app within a Docker container in a Docker-only environment, we then need to make a change to our existing `docker-compose.yaml`

Here we will need to specify AUTH_SERVICE_SERVICE_HOST as an actual environment variable (since Docker will not autogenerate it for us), and then give it the name of the service that we wish to contact (in this case `auth`).

`docker-compose.yaml`

```
….
…..
  users:
    build: ./users-api
    environment:
      AUTH_ADDRESS: auth
      AUTH_SERVICE_SERVICE_HOST: auth
    ports:
      - "8080:8080"
  tasks:
….
….
```

7.3 Using CoreDNS for internal domain names of services

CoreDNS is a flexible and extensible DNS server that is responsible for providing service discovery and DNS resolution within a Kubernetes cluster. Kubernetes automatically creates DNS entries for services, pods, and other resources, which CoreDNS manages. CoreDNS then resolves internal domain names for services to their corresponding cluster IP address.

Kubernetes uses a specific convention for internal service domain names:

```
<service-name>.<namespace>.svc.cluster.local
```

Where:
`<service-name>`: The name of the service.
`<namespace>`: The namespace where the service resides.
`svc.cluster.local`: The default domain suffix for services within the cluster.

The default namespace for Kubernetes services is `default`, if none is explicitly specified.

Check again on the running services:

```
kubectl get services
```

Based on this convention, the `auth-service` would have an internal service domain name of:

```
auth-service.default.svc.cluster.local
```

We can start a pod based on a simple Linux image that gives us access to the `nslookup` Linux tool to verify that CoreDNS will be able to resolve this internal service domain name to the correct internal IP address (CLUSTER-IP) of that service.

```
kubectl run -it --rm dns-test --image=busybox --restart=Never --
nslookup auth-service.default.svc.cluster.local
```

You can repeat this DNS resolution to ClusterIP address for the `user-service` if you wish to.

We now again change `users-deployment.yaml` to use the internal domain name for the `auth-service`.

`users-deployment.yaml`

```

….
….
```

```
....
      containers:
        - name: users
          image: dockerhubaccount/kub-demo-users
          ports:
            - containerPort: 8080
          env:
            - name: AUTH_ADDRESS
              value: "auth-service.default.svc.cluster.local"
```

To incorporate this change, we just simply apply this YAML file again:

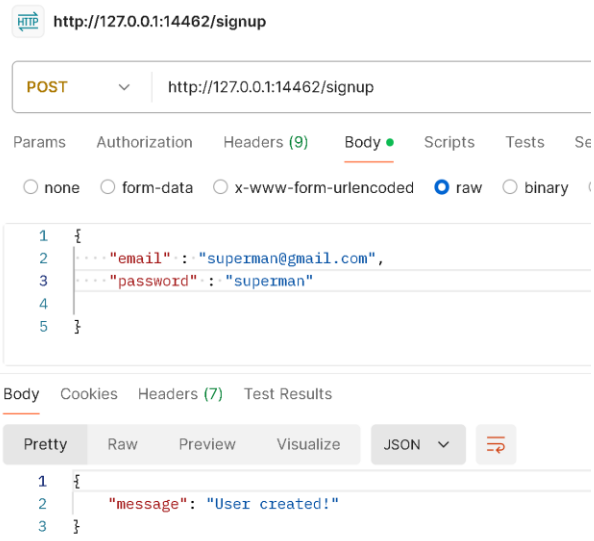
```
kubectl apply -f users-deployment.yaml
```

Finally, to test that everything is still working, we repeat the previous actions from a previous lab session:

Using the URL and port number exposed by the `users-service` for the `users-deployment` pod, send a POST request to the `user-api` app at the API endpoint `/signup` with a JSON content for a username/password pair

```
{
"email" : "superman@gmail.com",
"password" : "superman"
}
```

Verify that you can get back a successful message.
This indicates that the `user` app was able to successfully communicate with the `auth` app via the internal domain name for the `auth-service` that exposes the `auth` app.
Both apps are now running in two different pods within the same node, each with their own internal IP address.



POST http://127.0.0.1:14462/signup

```
{
    "email" : "superman@gmail.com",
    "password" : "superman"

}
```

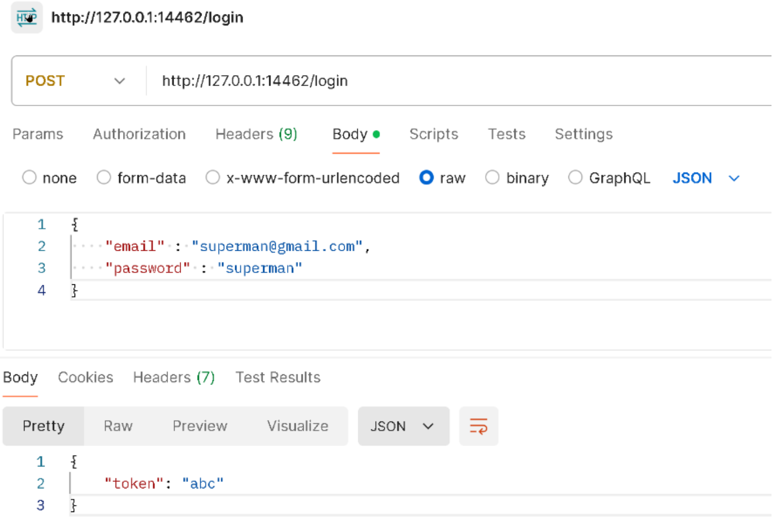
```
{
    "message": "User created!"
}
```

Just as before, you can verify the actual API endpoints that were called in both apps (`users` and `auth`) which is displayed in `console.log` statements and are available by examining the log output from these two pods: either through Minikube Dashboard UI or the VS Code Kubernetes extension or directly through a CLI command as explained previously.

In particular, you should see the GET request from the `users` app targeting the internal domain name that you specified for the `AUTH_ADDRESS` environment variable in `users-deployment.yaml`

```
The signup API endpoint invoked
Sending GET request to  http://auth-service.default.svc.cluster.local/hashed-password/superman
```

Verify that we will also get back a token if we send a POST request with similar JSON content for a username/password pair to the API endpoint: `/login`



```
{
    "email" : "superman@gmail.com",
    "password" : "superman"
}
```

```
{
    "token": "abc"
}
```

Just as before, you can verify the actual API endpoints that were called in both apps (`users` and `auth`) which is displayed in `console.log` statements and are available by examining the log output from these two pods: either through Minikube Dashboard UI or the VS Code Kubernetes extension or directly through a CLI command as explained previously.

Both the successful responses indicating that the code within the `user` app that handles these two API endpoints (`/signup` and `/login`) has successfully executed HTTP API calls to the `auth` running in a different pod on the same node, using the internal domain name for the service that exposes the `auth` app (`auth-service`)

In general, using domain names is the most common way to connect pods because they are easier to remember and implement.

8 Adding the Task app as a deployment

We will refactor `tasks-api\tasks-app.js` to allow it to incorporate an environment variable to allow it to work with both Kubernetes and Docker Compose tasks-app.js

```
…..

  if (!headers.authorization) {
    throw new Error('No token provided.');
  }
  const token = headers.authorization.split(' ')[1];
  // expects Bearer TOKEN whcih is the 2nd string after term Bearer

  console.log("Token extracted is : ",token);

  const authVerifyAPIEndpoint = `http://${process.env.AUTH_ADDRESS}/verify-
token/` + token;

  console.log("Making a call to verify the token to this API endpoint:
",authVerifyAPIEndpoint);

  const response = await axios.get(authVerifyAPIEndpoint);

  console.log("Got back a response with uid : ",response.data.uid);

  return response.data.uid;

…..
```

In a Powershell terminal in the `tasks-api` folder, build the image with a new tag:

```
docker build -t dockerhubaccount/kub-demo-tasks .
```

When the image is built, push it to your DockerHub account:

```
docker push dockerhubaccount/kub-demo-tasks
```

In `kubernetes` folder, create a deployment configuration and service configuration YAML for this `task-api` app.

tasks-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: tasks-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: tasks
  template:
    metadata:
      labels:
        app: tasks
    spec:
      containers:
        - name: tasks
          image: dockerhubaccount/kub-demo-tasks
          env:
            - name: AUTH_ADDRESS
              value: "auth-service.default.svc.cluster.local"
            - name: TASKS_FOLDER
              value: tasks
```

`tasks-service.yaml`

```
apiVersion: v1
kind: Service
metadata:
  name: tasks-service
spec:
  selector:
    app: tasks
  type: LoadBalancer
  ports:
    - protocol: TCP
      port: 8000
      targetPort: 8000
```

Apply both of these configuration manifests with:

```
kubectl apply -f tasks-service.yaml -f tasks-deployment.yaml
```

Check the pods and deployments available:

```
kubectl get deployments
```

```
kubectl get pods
```

You should now see 3 distinct deployments for the 3 distinct apps, each running an individual pod.

Now we check the services:

```
kubectl get services
```

to verify that the `tasks-service` is indeed up and running alongside the two other services: `auth-service` and `users-service`.

Once all the relevant services and deployments are ready, in a new PowerShell terminal, run the service with:

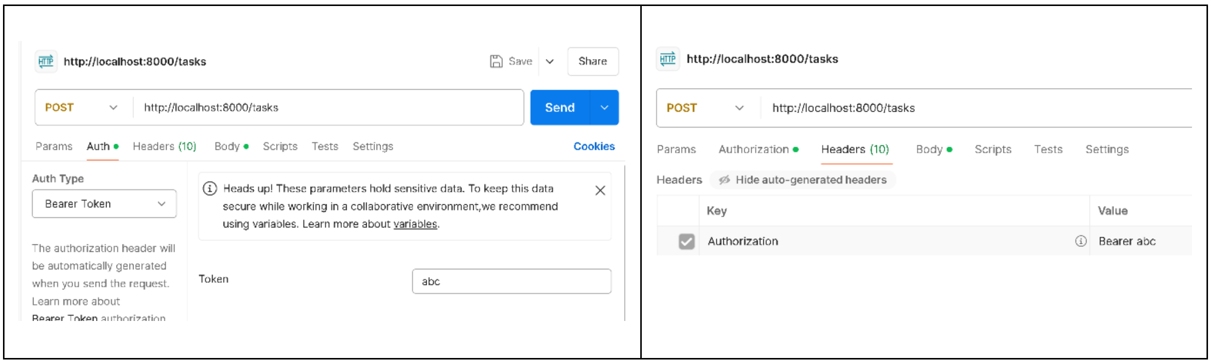
```
minikube service tasks-service
```

As we have already seen, Minikube provides a specific way to access the port on the NodePort: it creates a tunnel from a port on localhost to the NodePort. This localhost port is randomly assigned from the range of free ports on the local machine. You can then use the loopback address (`127.0.0.1`) or `localhost` at this tunnelling port to forward HTTP traffic to `tasks-deployment`.

Using the tunnelling port number for the `tasks-service`, we can now send a POST request to the `task-api` app at the API endpoint `/tasks` with a JSON content for a new task (which will be a combination of text and title).

Just as in the case in the original scenario, where we ran all 3 apps as containers which interacted with each other in a purely Docker environment, we need to first use the token we obtained earlier from the `users` app as an Authorization token in the header of all HTTP requests that we now sent to the `tasks` app

As before in that original scenario, you will first need to configure the dummy token (`abc`) as an Authorization token in an additional header to be sent out in all the HTTP requests sent out to the service that exposes this app (tasks-service). You can set this in the Authorization section of the Request page on Postman, using Auth Type as Bearer Token and setting `abc` as the token value. You should then be able to view this as an additional header in the Headers section.



http://localhost:8000/tasks

POST http://localhost:8000/tasks

Auth Type: Bearer Token

Heads up! These parameters hold sensitive data. To keep this data secure while working in a collaborative environment,we recommend using variables. Learn more about variables.

The authorization header will be automatically generated when you send the request. Learn more about Bearer Token authorization

Token: abc

http://localhost:8000/tasks

POST http://localhost:8000/tasks

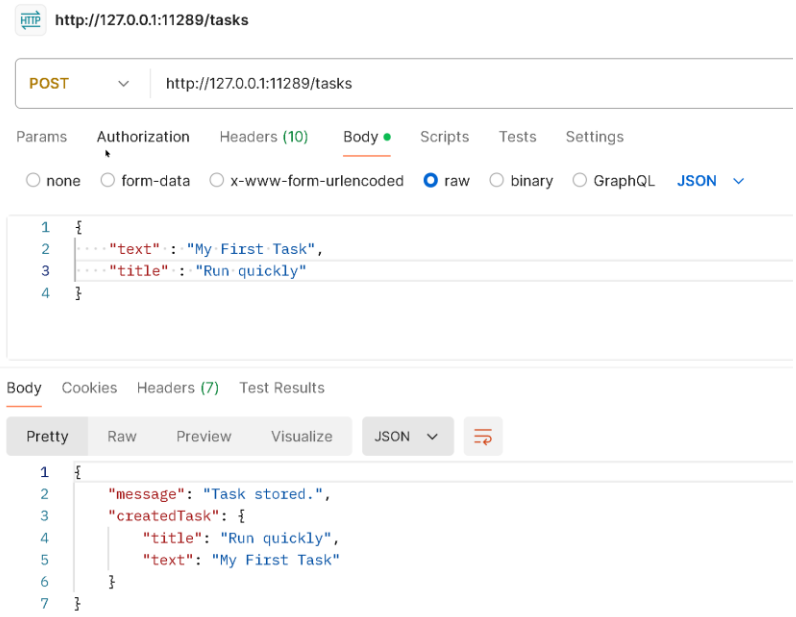
Headers

	Key	Value
☑	Authorization	Bearer abc

Once this is done, you can configure the Body section some dummy JSON data to be stored, for e.g.

```
{
    "text" : "My First Task",
    "title" : "Fly in the Sky"
}
```

You can then send out a POST request with some dummy data to be stored as JSON content for a text / title combination. The app should return a response indicating successful retrieval of the token.



http://127.0.0.1:11289/tasks

POST http://127.0.0.1:11289/tasks

```
{
    "text" : "My First Task",
    "title" : "Run quickly"
}
```

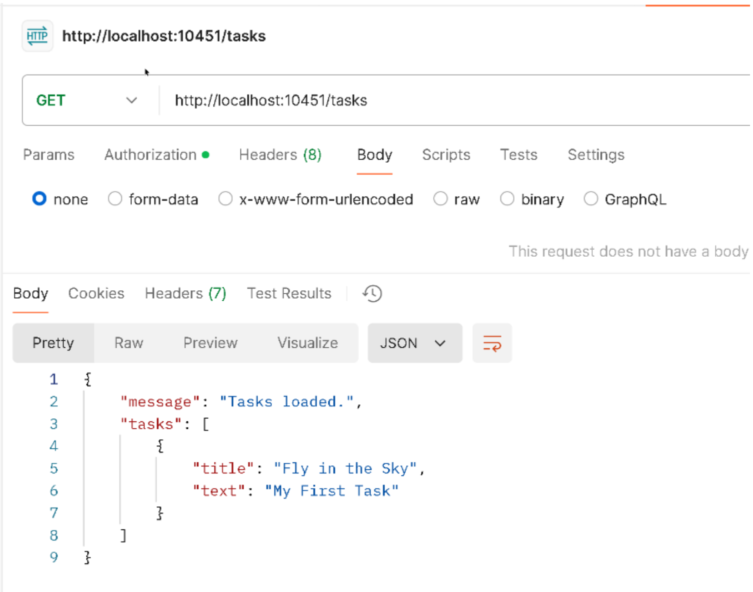
```
{
    "message": "Task stored.",
    "createdTask": {
        "title": "Run quickly",
        "text": "My First Task"
    }
}
```

Just as before, you can verify the actual API endpoints that were called in both apps (`tasks` and `auth`) which is displayed in `console.log` statements and are available by examining the log output from these two pods: either through Minikube Dashboard UI or the VS Code Kubernetes extension or directly through a CLI command as explained previously.

In particular, you should see the GET request from the `tasks` app targeting the internal domain name that you specified for the `AUTH_ADDRESS` environment variable in `tasks-deployment.yaml`

```
Making a call to verify the token to this API endpoint: http://auth-service.default.svc.cluster.local/verify-token/abc
```

Finally, you can send a GET request to the same API endpoint to retrieve the stored task. You need to do sent the GET request in the same tab in Postman, or else if you wish to do this in a new tab, you will need to configure the Authorization token of `abc` again.



```json
{
    "message": "Tasks loaded.",
    "tasks": [
        {
            "title": "Fly in the Sky",
            "text": "My First Task"
        }
    ]
}
```

You can continue to send POST requests with further data for title / text JSON content to this API endpoint and subsequently retrieve the list of tasks so far with a GET request to the same endpoint.

Just as before, you can verify the actual API endpoints that were called in both apps (`tasks` and `auth`) which is displayed in `console.log` statements and are available by examining the log output from these two pods: either through Minikube Dashboard UI or the VS Code Kubernetes extension or directly through a CLI command as explained previously.

Finally, we can also modify `docker-compose.yaml` in the top level root project folder to provide a Docker specific value for this environment variable. Remember that in Compose project, the services will use the service name (in this case: `auth`) in order to communicate with each other in the application code.

`docker-compose.yaml`

```yaml
....
....

  tasks:
    build: ./tasks-api
    ports:
      - "8000:8000"
    environment:
      TASKS_FOLDER: tasks
      AUTH_ADDRESS: auth
```

9 END