Containers and DevOps

Hands-on lab step-by-step

February 2018

Information in this document, including URL and other Internet Web site references, is subject to change without notice. Unless otherwise noted, the example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, e-mail address, logo, person, place or event is intended or should be inferred. Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

The names of manufacturers, products, or URLs are provided for informational purposes only and Microsoft makes no representations and warranties, either expressed, implied, or statutory, regarding these manufacturers or the use of the products with any Microsoft technologies. The inclusion of a manufacturer or product does not imply endorsement of Microsoft of the manufacturer or product. Links may be provided to third party sites. Such sites are not under the control of Microsoft and Microsoft is not responsible for the contents of any linked site or any link contained in a linked site, or any changes or updates to such sites. Microsoft is not responsible for webcasting or any other form of transmission received from any linked site. Microsoft is providing these links to you only as a convenience, and the inclusion of any link does not imply endorsement of Microsoft of the site or the products contained therein.

© 2018 Microsoft Corporation. All rights reserved.

Microsoft and the trademarks listed at <https://www.microsoft.com/en-us/legal/intellectualproperty/Trademarks/Usage/General.aspx> are trademarks of the Microsoft group of companies. All other trademarks are property of their respective owners.

Contents

[Containers and DevOps hands-on lab step-by-step 1](#_Toc505885604)

[Abstract and learning objectives 1](#_Toc505885605)

[Overview 1](#_Toc505885606)

[Requirements 1](#_Toc505885607)

[Before the hands-on lab 3](#_Toc505885608)

[Task 1: Explore Pre-Configured Environment 4](#_Toc505885609)

[Exercise 1: Environment setup 5](#_Toc505885610)

[Task 1: Verify the FabMedical starter files 6](#_Toc505885611)

[Exercise 2: Create and run a Docker application 11](#_Toc505885612)

[Task 1: Test the application 11](#_Toc505885613)

[Task 2: Enable browsing to the web application 12](#_Toc505885614)

[Task 3: Create a Dockerfile 14](#_Toc505885615)

[Task 4: Create Docker images 16](#_Toc505885616)

[Task 5: Run a containerized application 17](#_Toc505885617)

[Task 6: Setup environment variables 18](#_Toc505885618)

[Task 7: Push images to Azure Container Registry 20](#_Toc505885619)

[Exercise 3: Deploy the solution to Azure Container Service 23](#_Toc505885620)

[Task 1: Tunnel into the Azure Container Service cluster 23](#_Toc505885621)

[Task 2: Deploy a service using the Kubernetes management dashboard 25](#_Toc505885622)

[Task 3: Deploy a service using Kubernetes REST API 32](#_Toc505885623)

[Task 4: Explore service instance logs and resolve an issue 37](#_Toc505885624)

[Task 5: Test the application in a browser 43](#_Toc505885625)

[Exercise 4: Scale the application and test HA 46](#_Toc505885626)

[Task 1: Increase service instances from the Kubernetes dashboard 46](#_Toc505885627)

[Task 2: Increase service instances beyond available resources 48](#_Toc505885628)

[Task 3: Restart containers and test HA 50](#_Toc505885629)

[Exercise 5: Setup load balancing and service discovery 56](#_Toc505885630)

[Task 1: Create a public load balancer for a service 56](#_Toc505885631)

[Task 2: Scale a service without port constraints 58](#_Toc505885632)

[Task 3: Update a service to support dynamic service discovery without a load balancer 60](#_Toc505885633)

[Task 4: Update an external service to support dynamic discovery with a load balancer 61](#_Toc505885634)

[Task 5: Adjust CPU constraints to improve scale 63](#_Toc505885635)

[Task 6: Perform a rolling update 64](#_Toc505885636)

Containers and DevOps hands-on lab step-by-step

## Abstract and learning objectives

Build a PoC to deliver a multi-tenant web app hosting solution leveraging Azure Container Service with Kubernetes, Docker containers, and Linux nodes.

Attendees will be better able to deploy Docker-based applications and scale them with Azure Container Service and Kubernetes orchestration. In addition,

* Create & run a Docker Application
* Deploy to the Azure Container Service with Kubernetes orchestration
* Implement load balancing and service discovery
* Scale the application and test availability

## Overview

Fabrikam Medical Conferences (FabMedical) provides conference website services tailored to the medical community. They are refactoring their application code, based on node.js, so that it can run as a Docker application, and want to implement a POC that will help them get familiar with the development process, lifecycle of deployment, and critical aspects of the hosting environment. They will be deploying their applications to Azure Container Service based on Kubernetes and want to learn how to deploy containers in a dynamically load-balanced manner, discover containers, and scale them on demand.

In this hands-on lab, you will assist with completing this POC with a subset of the application code base. You will work on a build agent based on Linux, and an Azure Container Service cluster for running deployed applications. You will be helping them to complete the Docker setup for their application, test locally, push to an image repository, deploy to the cluster, and test load-balancing and scale.

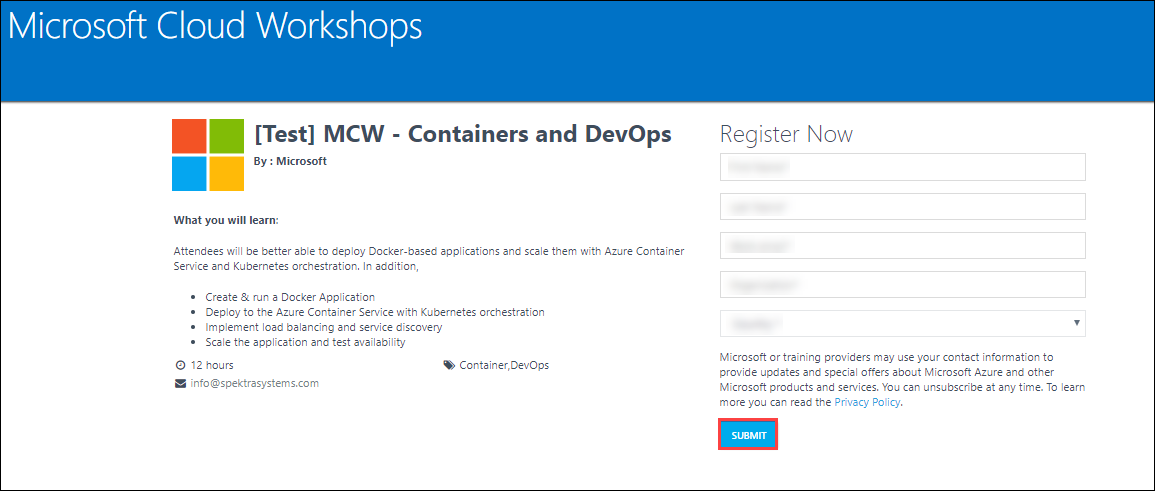
## Requirements

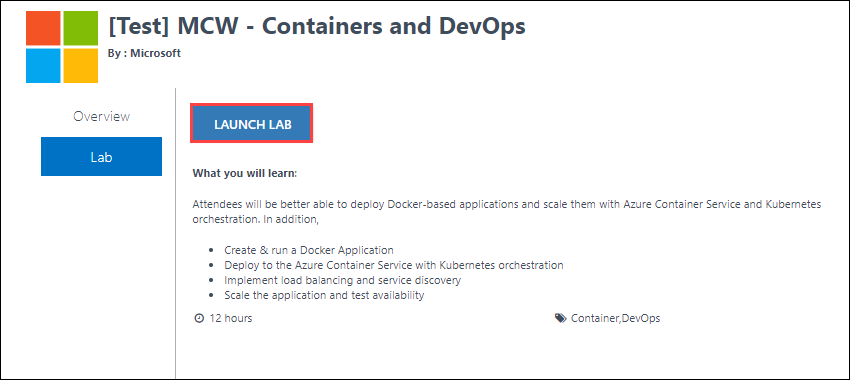
1. Local machine or a virtual machine configured with:
   1. A browser, preferably Chrome for consistency with the lab implementation tests
   2. SSH Client
      1. On Windows, you can use Putty or any other client. You can download Putty from here : <https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html>
      2. On Mac or Linux, all instructions should be executed using bash in Terminal.

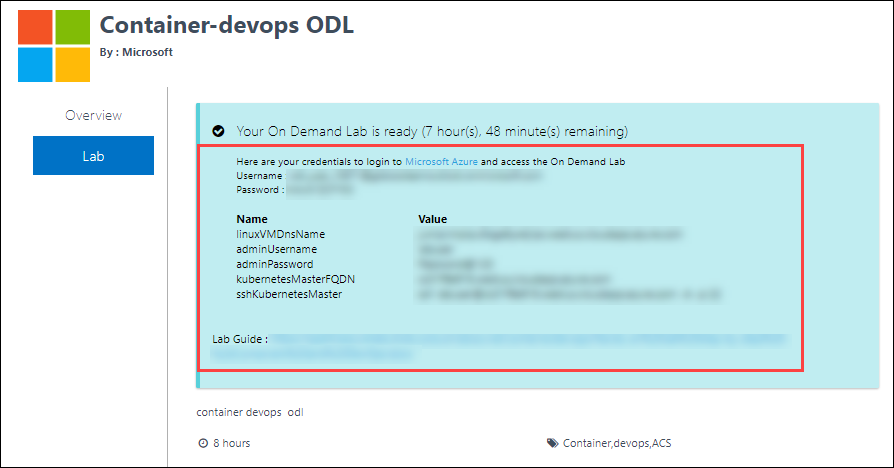
**VERY IMPORTANT: You should be typing all of the commands as they appear in the guide, except where explicitly stated in this document. Do not try to copy and paste from Word to your command windows or other documents where you are instructed to enter information shown in this document. There can be issues with Copy and Paste from Word that result in errors, execution of instructions, or creation of file content.**

## Before the hands-on lab

**Duration**: 10 mins

You should have signed-up for a pre-configured environment by navigating to the bit.ly URL provided to you at the venue by the workshop lead.   
Once you sign-up for the environment, Click on launch Lab button.

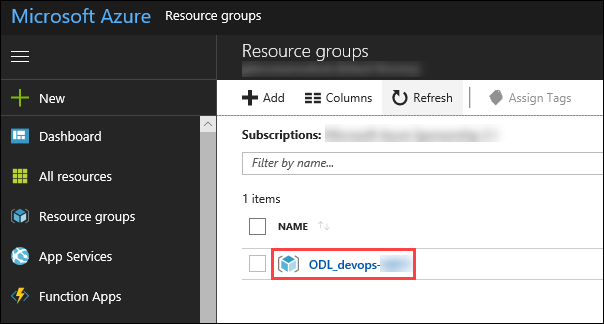
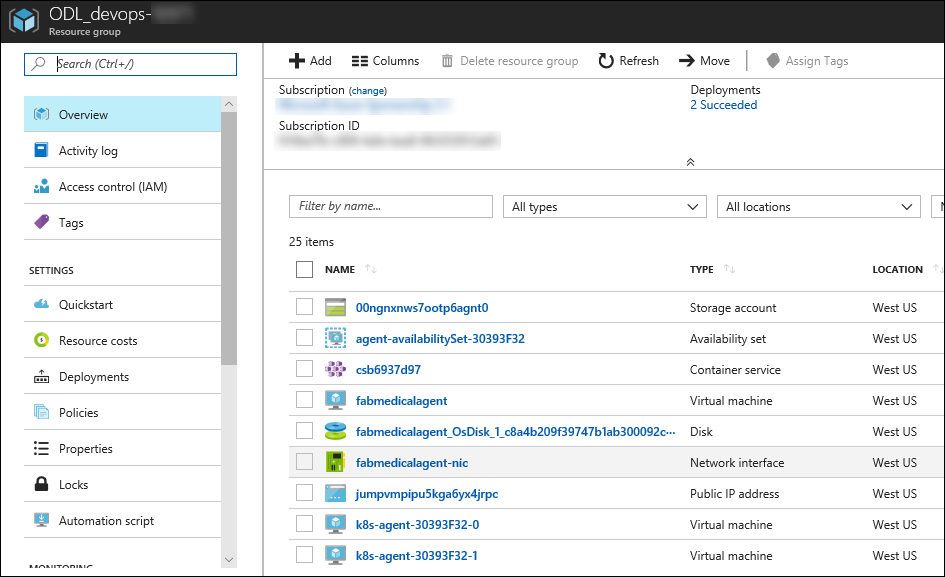


After some time, you will get credentials to access an Azure Environment and the credentials to login to the build agent VM.   


Please copy and save all the credentials assigned to your deployment.

You should follow all of the steps provided in this section *before* taking part in the hands-on lab ahead.

#### Task 1: Explore Pre-Configured Environment

1. Open a browser instance in InPrivate / incognito mode and navigate to https://portal.azure.com
2. Login to the portal using Azure Credentials issued for your environment.
3. Once you are logged in to the portal, navigate to **Resource Groups**.  
   
4. Note that you have access to one resource group   
   
5. Navigate to the resource group and view the already existing resources such as Container Service, Virtual Machines, etc.  
   

## Exercise 1: Environment setup

**Duration**: 10 minutes

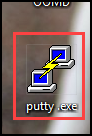
FabMedical has provided starter files for you. They have taken a copy of one of their websites, for their customer Contoso Neuro, and refactored it from a single node.js site into a website with a content API that serves up the speakers and sessions. This is a starting point to validate the containerization of their websites. They have asked you to use this to help them complete a POC that validates the development workflow for running the website and API as Docker containers and managing them within the Azure Container Service Kubernetes environment.

**NOTE:**

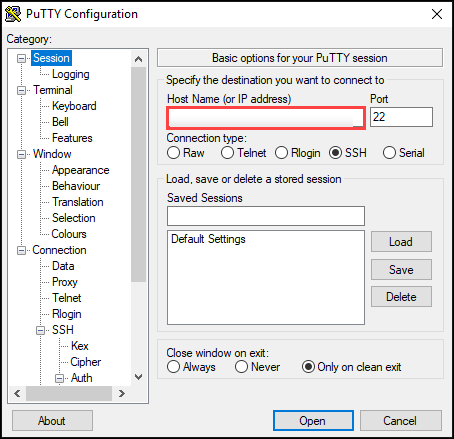
**Complete these tasks from the build agent Linux Vm session.**If you are using a Windows machine, you would need a **SSH** **client** for connecting to a Linux Virtual Machine. **Putty** is the most widely used SSH client for windows. If you are using Mac or Linux based machines, you can use the bash terminal.

#### Task 1: Verify the FabMedical starter files

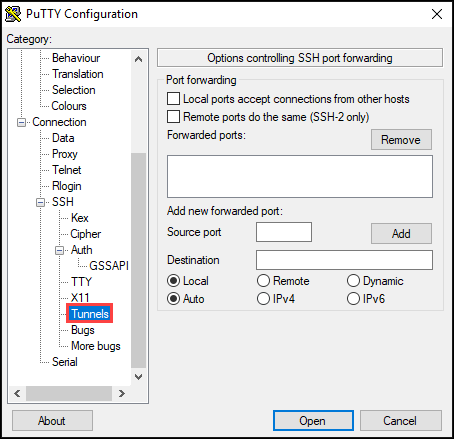
1. Connect to the build agent VM using the SSH client and the credentials you received earlier. If you are using putty to login to the Linux VM, go to step 2. Else If you are using a Linux terminal or Mac bash terminal or Git Client for Windows, go to step 14
2. Now **Download** a SSH Client from here, if you don’t already have one. <http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html>
3. Now run **putty.exe** from you PC.

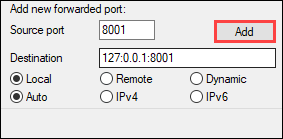


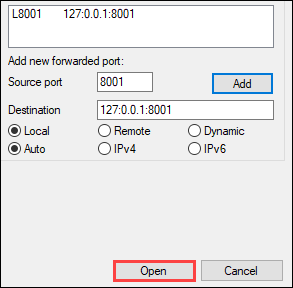
1. This is the application window that pops up when you run **putty.exe.**



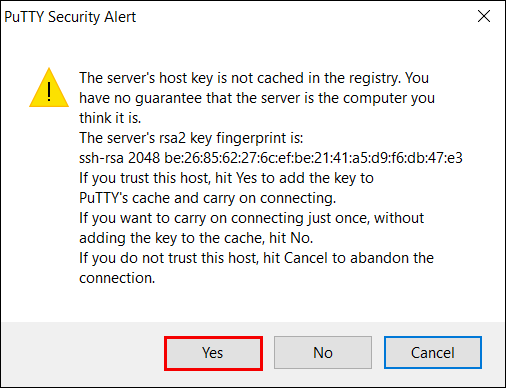
1. **Enter** the **linuxVMDnsName** of the VM to the **Host Name (or IP address)** box of the putty. Port will be **22** by default.
2. Go to the Tunnels section in PuTTY,



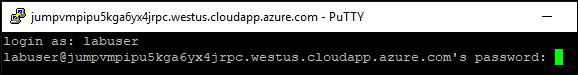
1. Now, you will configure a specific Local port 8001, that will redirect to 8001 of the build agent Linux VM.
2. Provide the following details and click on Add:  
   Source Port: **8001**  
   Destination: **127.0.0.1:8001**  
   Then **Click** on **Open.**

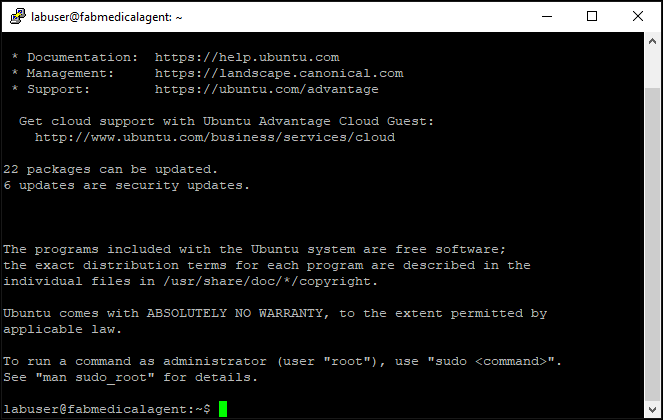


1. Now a new terminal will pop and you will be connected to the build agent virtual machine.
2. The **PuTTY Security Alert** will pop up. **Click** on **Yes.**

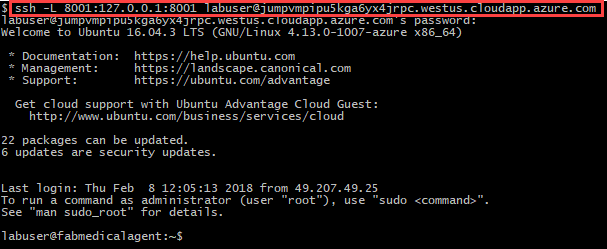


1. **Login** using the **adminUsername** and **adminPassword** for the **build agent Linux VM.**



1. After entering the username and password you can start accessing the **build agent Linux VM.**
2. Now go to step 15
3. For Mac or Linux or git client( Windows), use the following command to SSH into the build agent VM

ssh -L 8001:127.0.0.1:8001 labuser@[linuxVmDnsName]



1. Navigate to FabMedical folder and list the contents.
   * 1. cd FabMedical
     2. ll
2. You’ll see the listing includes two folders, one for the web site and another for the content API.
   * 1. content-api/
     2. content-web/
3. **NOTE: Keep this window open as your build agent SSH connection. You will later open new SSH sessions to other machines.**

## Exercise 2: Create and run a Docker application

**Duration**: 40 minutes

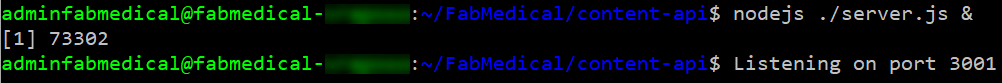
In this exercise, you will take the starter files and run the node.js application as a Docker application. You will create a Dockerfile, build Docker images, and run containers to execute the application.

**NOTE: Complete these tasks from the build agent Linux Vm session.**

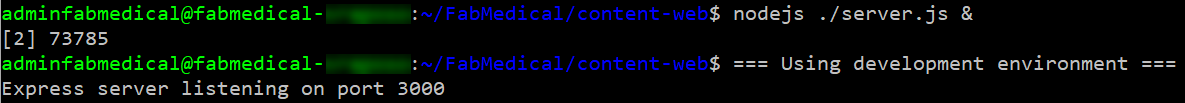
#### Task 1: Test the application

The purpose of this task is to make sure you can run the application successfully before applying changes to run it as a Docker application.

1. From the build agent vm terminal, navigate to the content-api directory and run npm install.
   * 1. cd content-api
     2. npm install
2. Start the API as a background process.
   * 1. nodejs ./server.js &



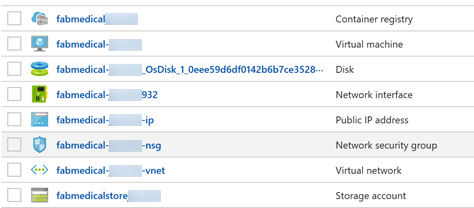
1. Press ENTER again to get to a command prompt for the next step.
2. Test the API using curl. You will request the speakers content, and this will return a JSON result.
   * 1. curl http://localhost:3001/speakers
3. Navigate to the web application directory, run npm install, and then run the application as a background process as well. Ignore any warnings you see in the output; this will not affect running the application.
   * 1. cd ..
     2. cd content-web
     3. npm install
     4. nodejs ./server.js &

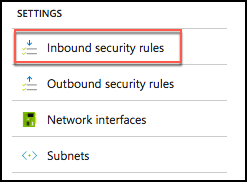


1. Press ENTER again to get a command prompt for the next step.
2. Test the web application using curl. You will see HTML output returned, without errors.
   * 1. curl http://localhost:3000
3. Leave the application running for the next task.
4. If you received a JSON response to the /speakers content request and an HTML response from the web application, your environment is working as expected.

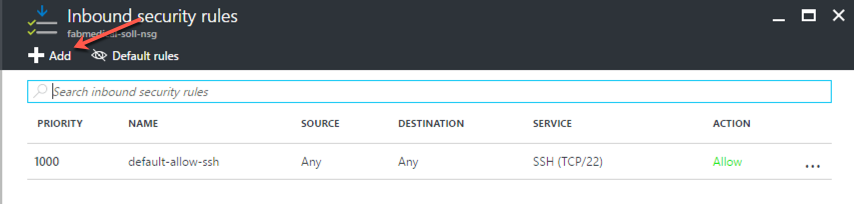
#### Task 2: Enable browsing to the web application

In this task, you will open a port range on the agent VM so that you can browse to the web application for testing.

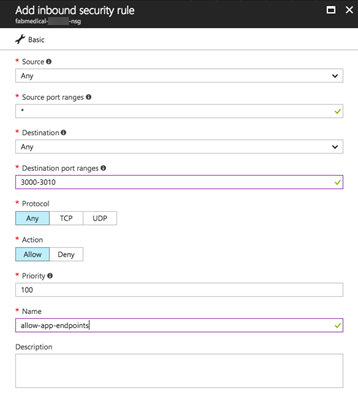
1. From the Azure portal select the resource group you have access to.
2. Select the Network Security Group named “**nsg**” associated with the build agent from your list of available resources. 
3. From the Network interface essentials blade, select **Inbound security rules**.

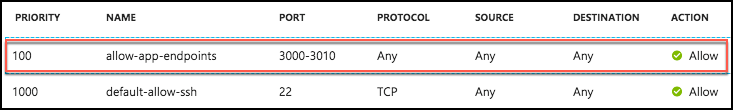


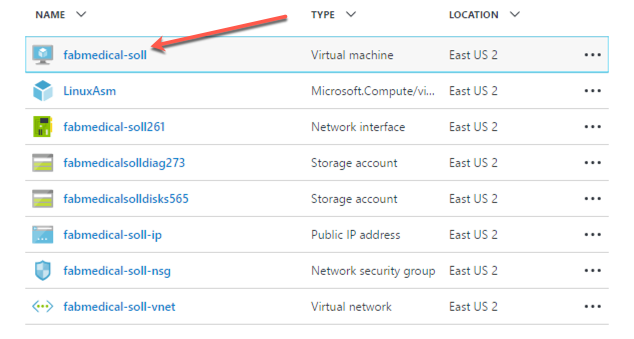
1. Select **Add** to add a new rule.

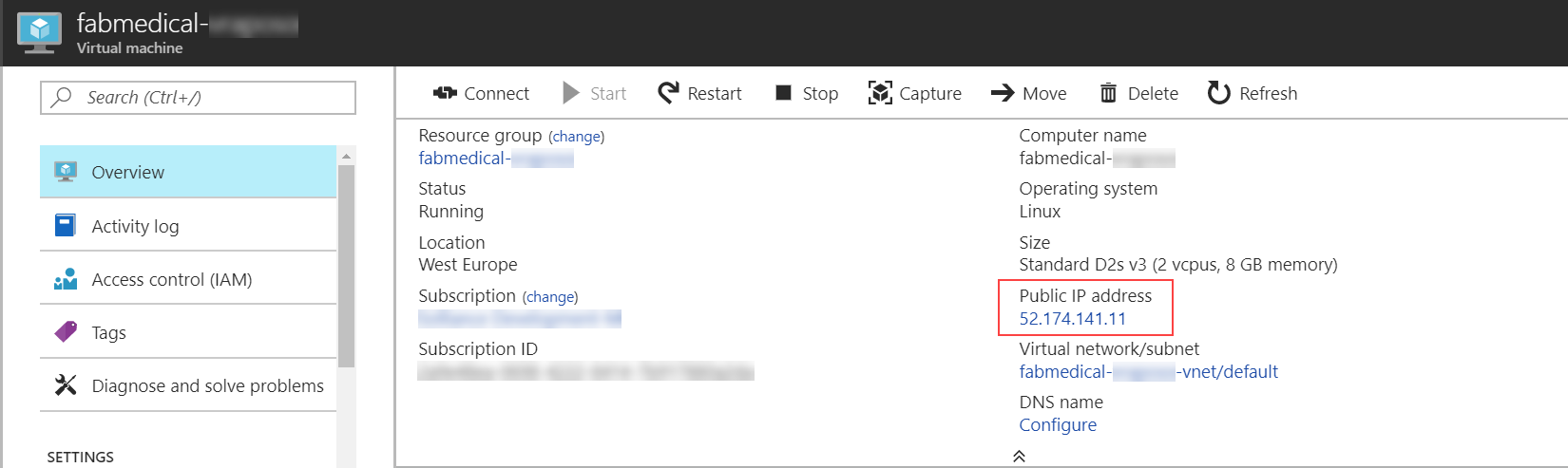


1. From the Add inbound security rule blade, enter the values as shown in the screenshot below:
   1. **Source**: Any
   2. **Source port ranges**: \*
   3. **Destination**: Any
   4. **Destination Port Ranges**: 3000-3010
   5. **Protocol**: Any
   6. **Action**: Allow
   7. **Priority**: Leave at the default priority setting
   8. **Name**: Enter “allow-app-endpoints”



1. Select **OK** to save the new rule. 
2. From the resource list shown in step 2, select the build agent VM named “**fabmedicalagent**”.



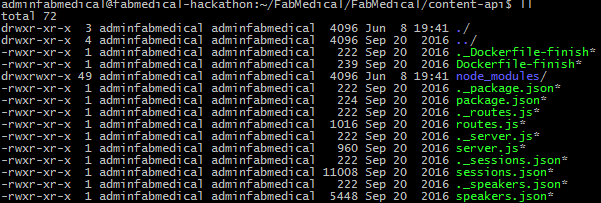
1. From the Virtual Machine blade overview, find the IP address of the VM. 
2. Test the web application from a browser. Navigate to the web application using your build agent IP address at port 3000.
   * 1. http://[BUILDAGENTIP]:3000
     2. EXAMPLE: http://13.68.113.176:3000
3. Select the Speakers and Sessions links in the header. You will see the pages display the HTML version of the JSON content you curled previously.
4. Once you have verified the application is accessible through a browser, go to your Build Agent Linux VM window and stop the running node processes.
   * 1. killall nodejs

#### Task 3: Create a Dockerfile

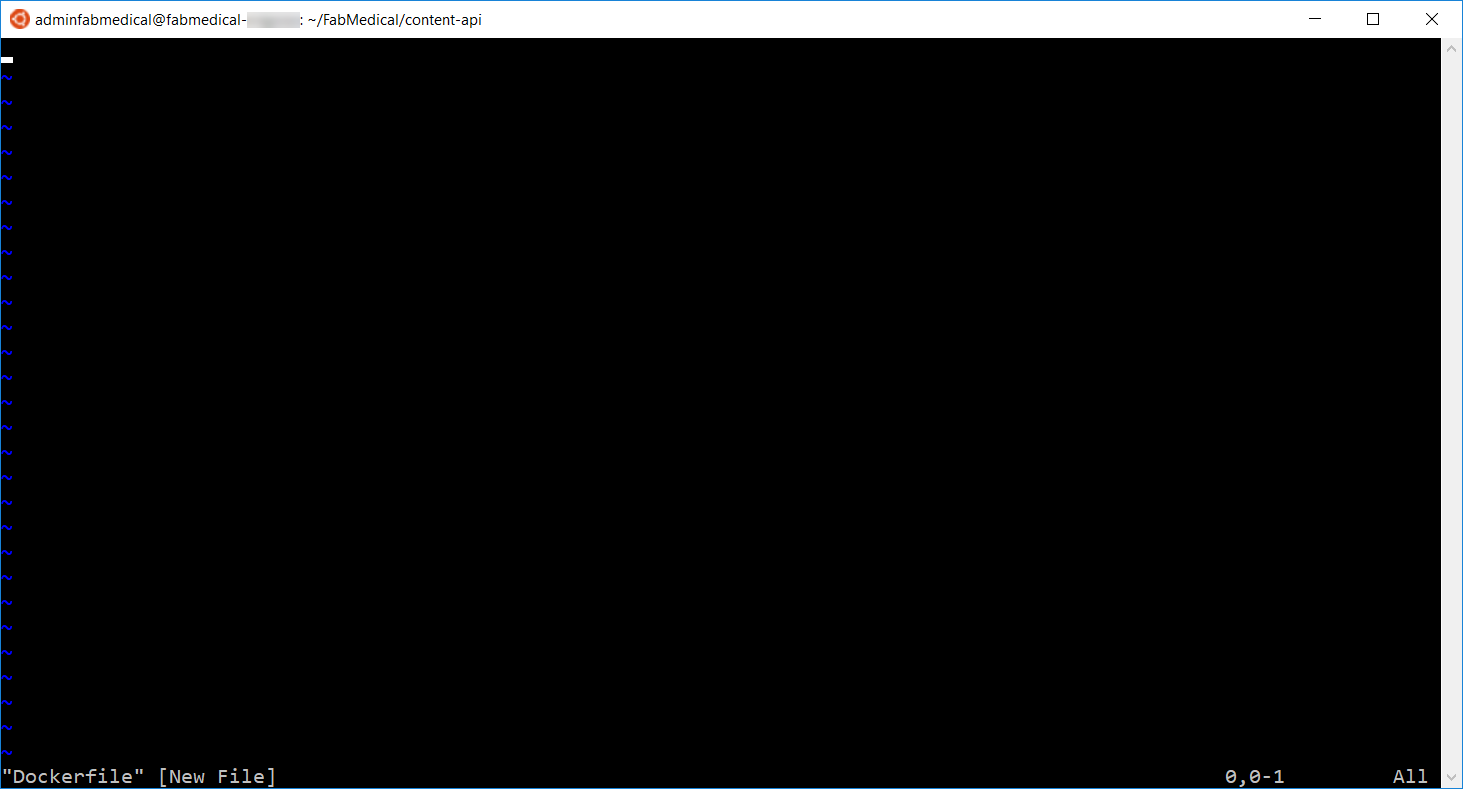
In this task, you will create a new Dockerfile that will be used to run the API application as a containerized application.

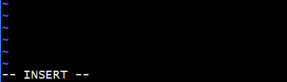
**NOTE: You will be working in a Linux VM without friendly editor tools. You must follow the steps very carefully to work with Vim for a few editing exercises, if you are not already familiar with Vim.**

1. From the Linux VM terminal, navigate to the content-api folder. List the files in the folder with this command. The output should look like the screenshot below.
   * 1. cd ..
     2. cd content-api
     3. ll



1. Create a new file named “Dockerfile” and note the casing in the name. Use the following Vim command to create a new file. The Build Agent VM Linux terminal window should look as shown in the following screenshot.
   * 1. vi Dockerfile

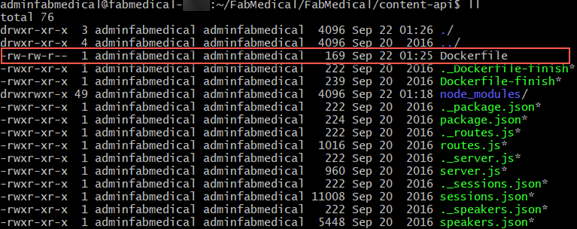


1. Select “i” on your keyboard. You’ll see the bottom of the window showing INSERT mode.
2. Type the following into the file. These statements produce a Dockerfile that describes the following:
   1. Creates a new Docker image from the base image node:argon. This base image has node.js on it and is compatible with the Ubuntu distribution of Linux you are deploying to.
   2. Creates a directory on the image where the application files can be copied.
   3. Copies the source files for the application over to the image.
   4. Runs npm install to initialize the node application environment.
   5. Exposes application port 3001 to the container environment so that the application can be reached at port 3001.
   6. Indicates the command to start the node application when the container is run.

**NOTE: Type the following into the editor, as you may have errors with copying and pasting.**

* + 1. FROM node:argon
    2. RUN mkdir -p /usr/src/app
    3. WORKDIR /usr/src/app
    4. COPY package.json /usr/src/app/
    5. RUN npm install
    6. COPY . /usr/src/app
    7. EXPOSE 3001
    8. CMD [ "npm", "start" ]

1. When you are finished typing, hit the Esc key and type “:wq” and hit the Enter key to save the changes and close the file.
   * 1. <Esc>
     2. :wq
     3. <Enter>
2. List the contents of the folder again, to verify the new Dockerfile has been created.
   * 1. ll

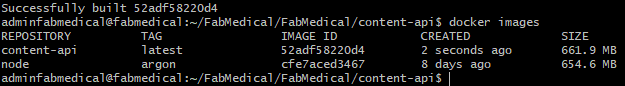


1. Verify the file contents, to ensure it was saved as expected. Type the following command to see the output of the Dockerfile in the command window.
   * 1. cat Dockerfile

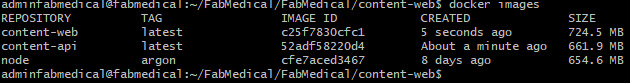
#### Task 4: Create Docker images

In this task, you will create Docker images for the application — one for the API application and another for the web application. Each image will be created via Docker commands that rely on a Dockerfile.

1. From Build agent VM Linux terminal, type the following command to view any Docker images on the VM. The list will be empty.
   * 1. docker images
2. From the content-api folder containing the API application files and the new Dockerfile you created, type the following command to create a Docker image for the API application. This command does the following:
   1. Executes the Docker build command to produce the image.
   2. Tags the resulting image with the name content-api (-t).
   3. The final dot (“.”) indicates to use the Dockerfile in this current directory context. By default, this file is expected to have the name “Dockerfile” (case sensitive).
      1. docker build -t content-api .
3. Once the image is successfully built, run the Docker images command again. You will see two images: the node image and your container image.
   * 1. docker images



1. Navigate to the content-web folder again and list the files. Note that this folder already has a Dockerfile.
   * 1. cd ..
     2. cd content-web
     3. ll
2. View the Dockerfile contents – which are similar to the file you created previously in the API folder. Type the following command:
   * 1. cat Dockerfile
3. Type the following command to create a Docker image for the web application.
   * 1. docker build –t content-web .
4. When complete, you will see three images now exist when you run the Docker images command.
   * 1. docker images



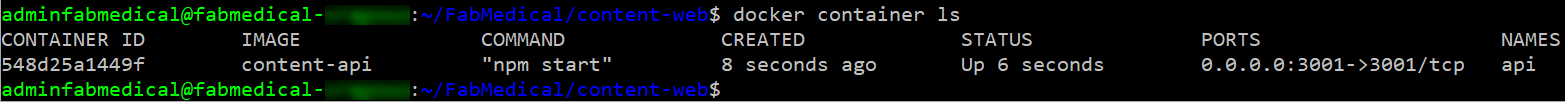
#### Task 5: Run a containerized application

The web application container will be calling endpoints exposed by the API application container. In this exercise, you will create a bridge network so that containers can communicate with one another, and then launch the images you created as containers in that network.

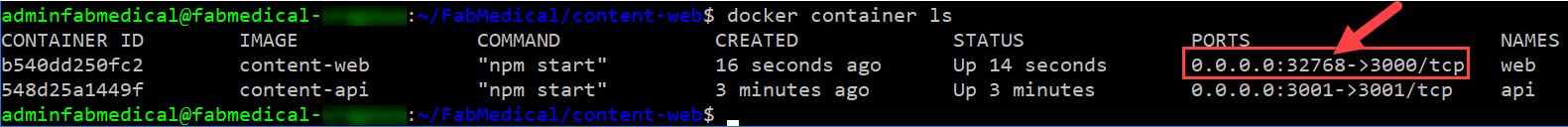
1. From build agent VM Linux terminal, type the following command to create a Docker network named “fabmedical”.
   * 1. docker network create fabmedical
2. Create and start the API application container with the following command. The command does the following:
   1. Creates a container from the specified image, by its tag, such as content-api.
   2. Names the container “api” for later reference with Docker commands.
   3. Issues the Docker run command indicating it should run the new container as a daemon.
   4. Instructs the Docker engine to use port 3001 and map that to the internal container port 3001.
   5. Instructs the Docker engine to use the “fabmedical” network.

docker run -d -p 3001:3001 --name api --net fabmedical content-api

1. Enter the command to show running containers. You’ll observe that the “api” container is in the list.
   * 1. docker container ls



1. Test the API by curling the URL. You will see JSON output as you did when testing previously.
   * 1. curl http://localhost:3001/speakers
2. Create and start the web application container with a similar Docker run command – instruct the docker engine to use any port with the -P command:
   * 1. docker run -d -P --name web --net fabmedical content-web
3. Enter the command to show running containers again and you’ll observe that both the API and web containers are in the list. The web container shows a dynamically assigned port mapping to its internal container port 3000.
   * 1. docker ps

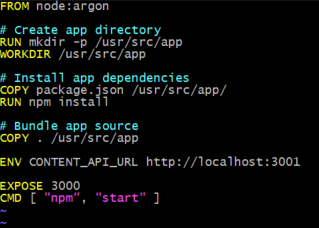


1. Test the web application by curling the URL. For the port, use the dynamically assigned port, which you can find in the output from the previous command. You will see HTML output, as you did when testing previously.
   * 1. :q

#### Task 6: Setup environment variables

In this task, you will configure the “web” container to communicate with the API container using an environment variable. You will modify the web application to read the URL from the environment variable, rebuild the Docker image, and then run the container again to test connectivity.

1. From build agent Linux VM windows, stop and remove the web container using the following commands.
   * 1. docker stop web
     2. docker rm web
2. Validate that the web container is no longer running or present by using the -a flag as shown in this command. You will see that the “web” container is no longer listed.
   * 1. docker container ls -a
3. Navigate to the content-web\data-access directory. From there, open the index.js file for editing using Vim and press the “i" key to go into edit mode.
   * 1. cd data-access
     2. vi index.js
     3. <i>
4. Locate the following TODO item and modify the code to comment the first line and uncomment the second. The result is that the contentApiUrl variable will be set to an environment variable.
   * 1. //TODO: Exercise 2 - Task 6 - Step 4
     2. //var contentApiUrl = “http://localhost:3001”;
     3. var contentApiUrl = process.env.CONTENT\_API\_URL;
5. Press the Escape key and type “:wq” and then press the Enter key to save and close the file.
   * 1. <Esc>
     2. :wq
     3. <Enter>
6. Navigate to the content-web directory. From there open the Dockerfile for editing using Vim and press the “i" key to go into edit mode.
   * 1. cd ..
     2. vi Dockerfile
     3. <i>
7. Locate the EXPOSE line shown below and add a line above it that sets the environment variable as shown in the screenshot.
   * 1. ENV CONTENT\_API\_URL http://localhost:3001



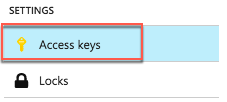
1. Press the Escape key and type “:wq” and then press the Enter key to save and close the file.
   * 1. <Esc>
     2. :wq
     3. <Enter>
2. Rebuild the web application Docker image using the same command as you did previously.
   * 1. docker build -t content-web .
3. Create and start the image passing the correct URI to the API container as an environment variable. This variable will address the API application using its container name over the Docker network you created. After running the container, check to see the container is running and note the dynamic port assignment for the next step.
   * 1. docker run -d -P --name web --net fabmedical -e CONTENT\_API\_URL=http://api:3001 content-web
     2. docker container ls
4. Curl the speakers path again, using the port assigned to the web container. This time you will see HTML returned.
   * 1. curl http://localhost:[PORT]/speakers.html
5. You will not be able to browse to the web application unless the agent VM exposes the port. Now you will stop the web container and restart it using port 3000 to test in the browser. To do this, type the following commands to stop the container, remove it, and run it again using explicit settings for the port.
   * 1. docker stop web
     2. docker rm web
     3. docker run -d -p 3000:3000 --name web --net fabmedical -e CONTENT\_API\_URL=http://api:3001 content-web
6. Curl the speaker path again, using port 3000. You will see the same HTML returned.
   * 1. curl http://localhost:3000/speakers.html
7. You can now use a web browser to navigate to the website and successfully view the application at port 3000. Replace [BUILDAGENTIP] with the IP address you used previously.
   * 1. http://[BUILDAGENTIP]:3000
     2. EXAMPLE: http://13.68.113.176:3000

#### Task 7: Push images to Azure Container Registry

To run containers in a remote environment, you will typically push images to a Docker registry, where you can store and distribute images. Each service will have a repository that can be pushed to and pulled from with Docker commands. Azure Container Registry (ACR) is a managed private Docker registry service based on Docker Registry v2.

In this task, you will push images to your ACR account, version images with tagging, and run containers from an image in your ACR.

1. In the [Azure Portal](https://portal.azure.com/), navigate to the ACR you created in Before the hands-on lab.
2. Select Access keys under Settings on the left-hand menu.



1. The Access keys blade displays the Login server, username, and password that will be required for the next step. Keep this handy as you perform actions on the build VM.

**NOTE: If the username and password do not appear, select Enable on the Admin user option.**

1. From the ssh session connected to your build Agent VM, login to your ACR account by typing the following command. Follow the instructions to complete the login.
   * 1. docker login [LOGINSERVER] –u [USERNAME] –p [PASSWORD]

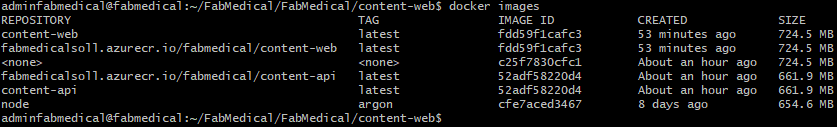
For example:

* + 1. docker login fabmedicalsoll.azurecr.io –u fabmedicalsoll –p +W/j=l+Fcze=n07SchxvGSlvsLRh/7ga

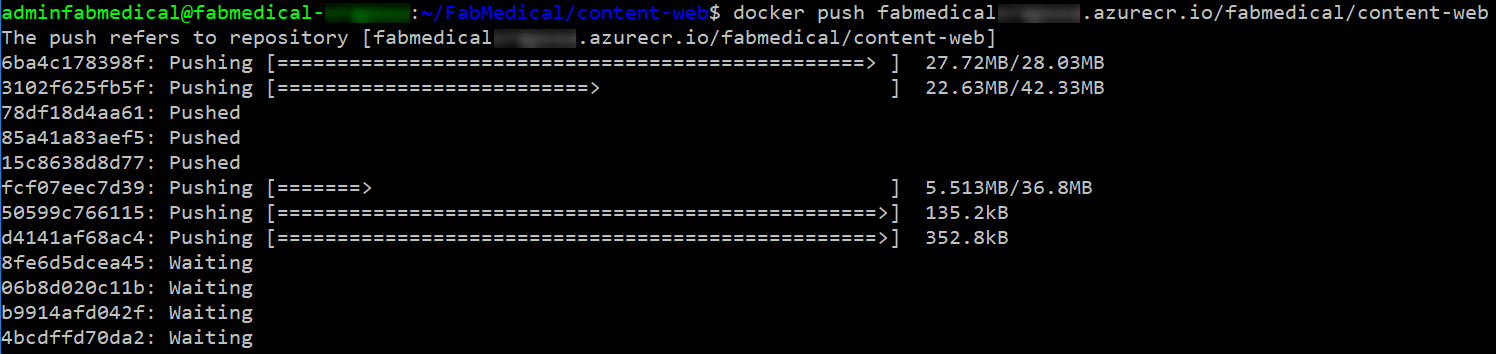


**Tip: Make sure to specify the fully qualified registry login server (all lowercase).**

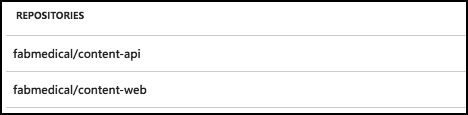
1. Run the following commands to properly tag your images to match your ACR account name. This also specifies the "fabmedical" namespace to avoid clutter in the root of the registry.
   * 1. docker tag content-web [LOGINSERVER]/fabmedical/content-web
     2. docker tag content-api [LOGINSERVER]/fabmedical/content-api
2. List your docker images and look at the repository and tag. Note that the repository is prefixed with your ACR account name, and fabmedical namespace, such as the sample shown in the screenshot below.
   * 1. docker images

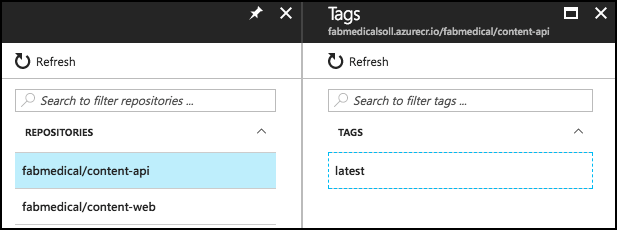


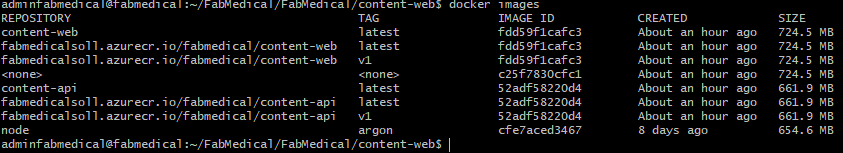
1. Push the images to your ACR account with the following command.
   * 1. docker push [LOGINSERVER]/fabmedical/content-web
     2. docker push [LOGINSERVER]/fabmedical/content-api



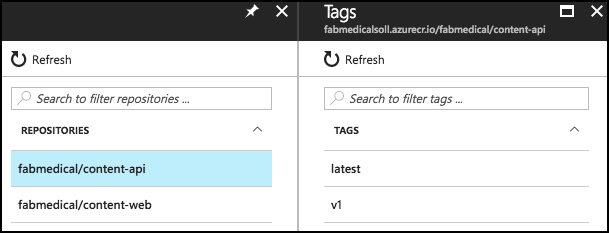
1. In the Azure Portal, navigate to your ACR account, and select Repositories under Services on the left-hand menu. You will now see two, one for each image.



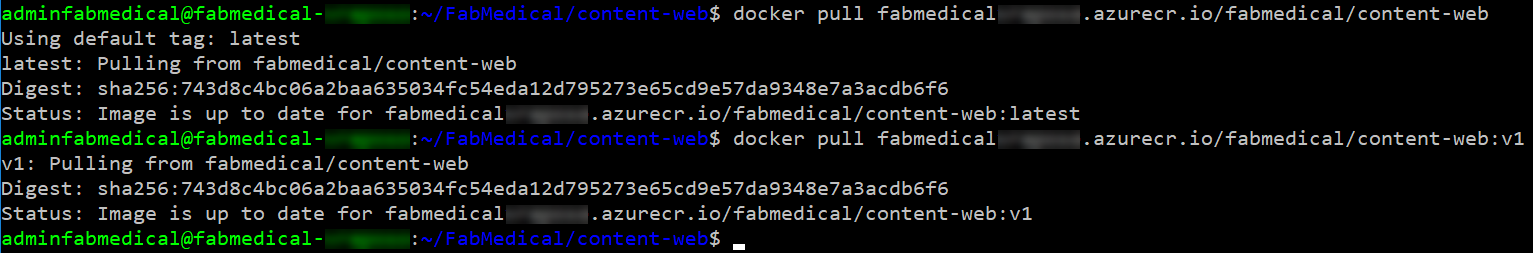
1. Select fabmedical/content-api. You’ll see the latest tag is assigned.
2. From build agent Linux VM Window, assign the v1 tag to each image with the following commands. Then, list the Docker images to note that there are now two entries for each image, showing the latest tag and the v1 tag. Also note that the image ID is the same for the two entries, as there is only one copy of the image.
   * 1. docker tag [LOGINSERVER]/fabmedical/content-web:latest [LOGINSERVER]/fabmedical/content-web:v1
     2. docker tag [LOGINSERVER]/fabmedical/content-api:latest [LOGINSERVER]/fabmedical/content-api:v1
     3. docker images



1. Repeat Step 7 to push the images to ACR again, so that the newly tagged v1 images are pushed. Then, refresh one of the repositories to see the two versions of the image now appear.



1. Run the following commands to pull an image from the repository. Note that the default behavior is to pull images tagged with “latest.” You can pull a specific version using the version tag. Also, note that since the images already exist on the build agent, nothing is downloaded.
   * 1. docker pull [LOGINSERVER]/fabmedical/content-web
     2. docker pull [LOGINSERVER]/fabmedical/content-web:v1



## Exercise 3: Deploy the solution to Azure Container Service

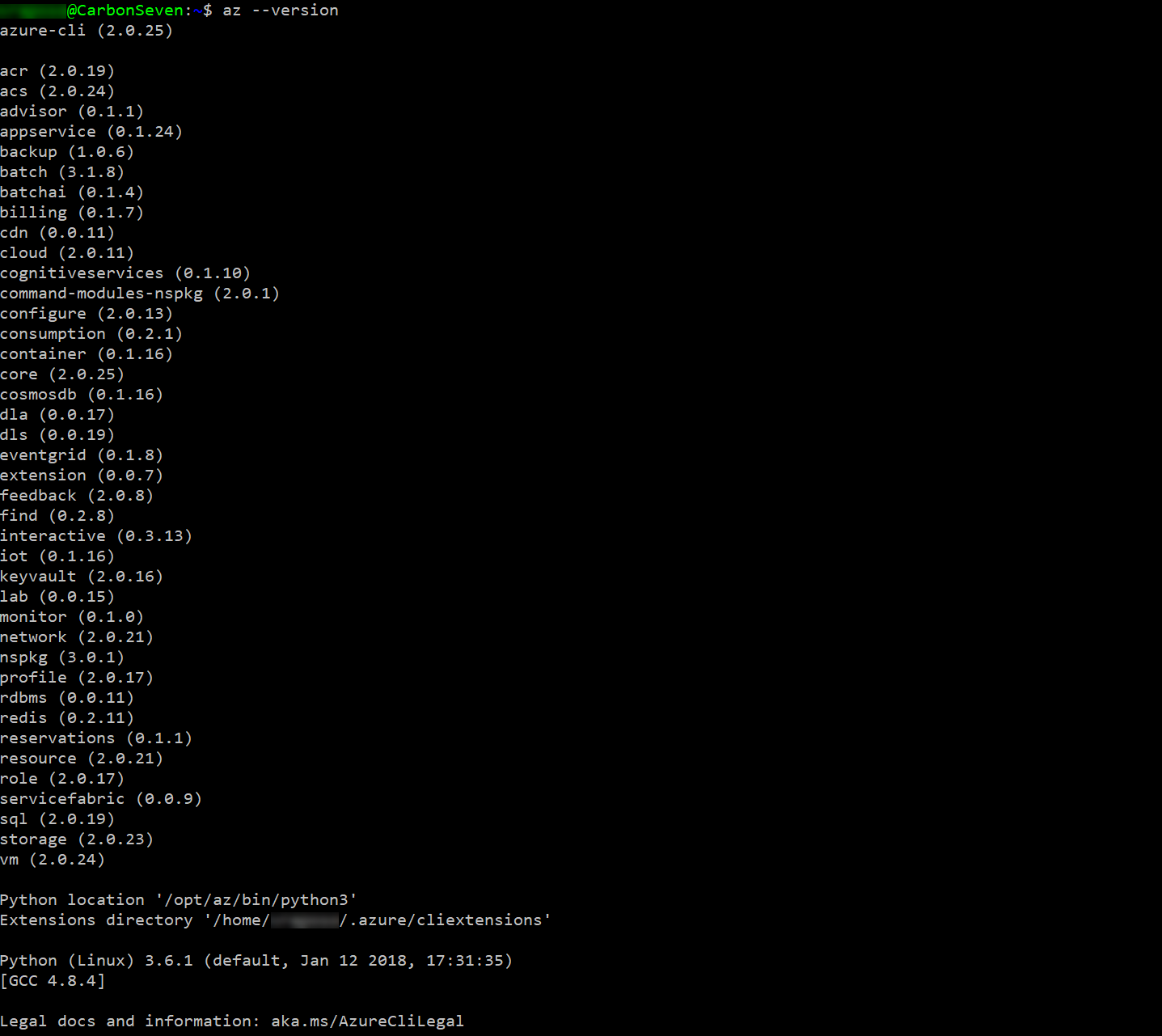
**Duration**: 30 minutes

In this exercise, you will connect to the Azure Container Service cluster you created before the hands-on lab and deploy the Docker application to the cluster using Kubernetes.

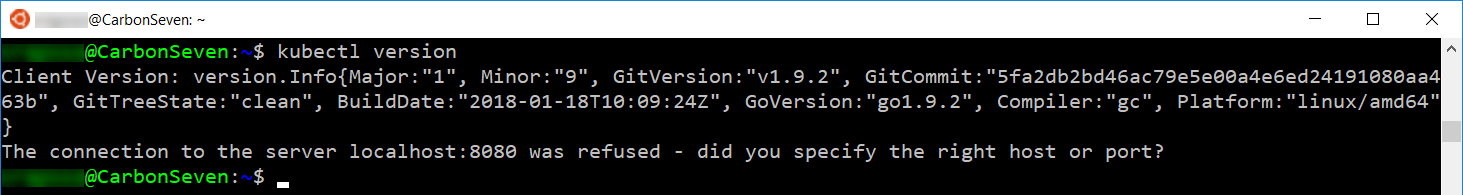
#### Task 1: Tunnel into the Azure Container Service cluster

In this task, you will gather the information you need about your Azure Container Service cluster to connect to the cluster and execute commands to connect to the Kubernetes management dashboard from your local machine.

1. Open the build agent VM terminal, ensure Azure CLI is installed correctly by running the following command:
   * 1. az --version
   1. This should produce output similar to this:



1. Also, check the installation of the Kubernetes CLI (kubectl) by running the following command:
   * 1. kubectl version
   1. This should produce output similar to this:



1. Once you have verified Azure CLI and Kubernetes CLI, login with the following command, following the instructions to complete your login as presented:
   * 1. az login
2. Verify that you are connected to the correct subscription with the following command to show your default subscription:

az account show

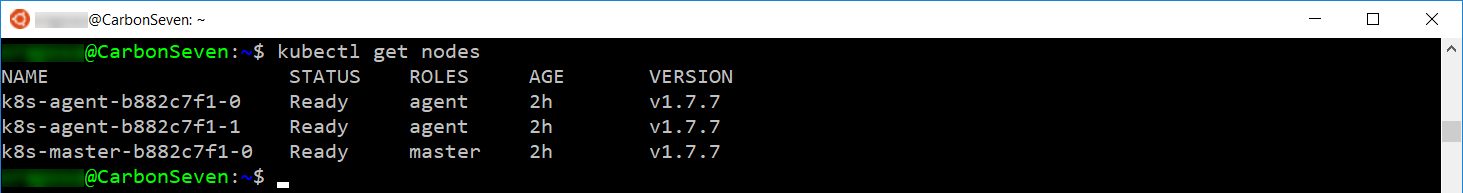
1. Move to the home directory by executing the following command:

cd

1. Configure kubectl to connect to the Kubernetes cluster. You must specify your id\_rsa ssh key.

az acs kubernetes get-credentials --resource-group=[resourcegroupname] --name=[ContainerServiceName] --ssh-key-file=.ssh/id\_rsa

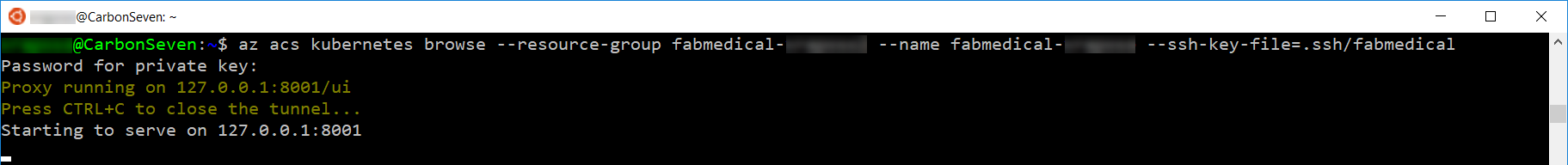
1. Test that the configuration is correct by running a simple kubectl command to produce a list of nodes:
   * 1. kubectl get nodes

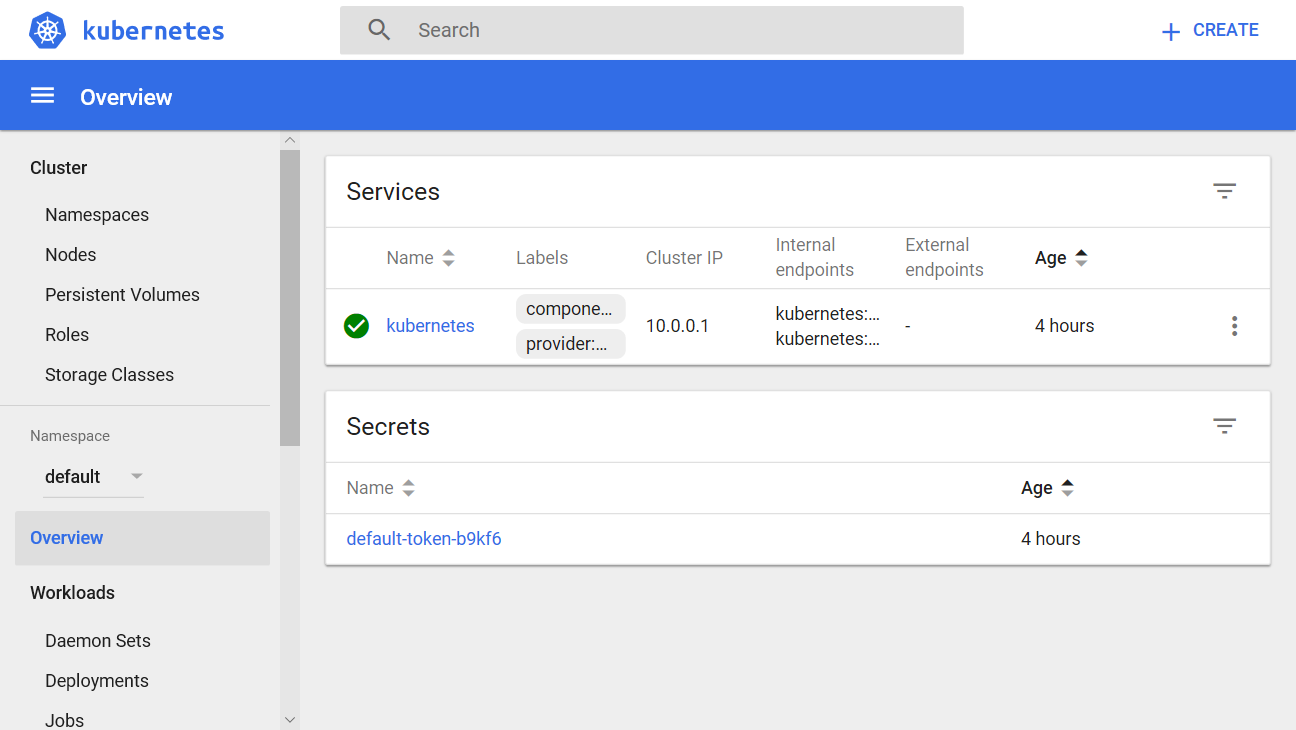


1. Create an SSH tunnel linking a local port (8001) on your machine to port 80 on the management node of the cluster. Execute the command below replacing the values as follows:
   1. Replace [PRIVATEKEY] with the location of your SSH key file, such as .ssh/id\_rsa
   2. Replace [RESOURCEGROUP] with the existing resource group name
   3. Replace [CONTAINERINSTANCE] with the container service instance name in the existing resource group

az acs kubernetes browse --resource-group [RESOURCEGROUP] --name [CONTAINERINSTANCE] --ssh-key-file=.ssh/id\_rsa

* 1. The output from this command will look similar to this:

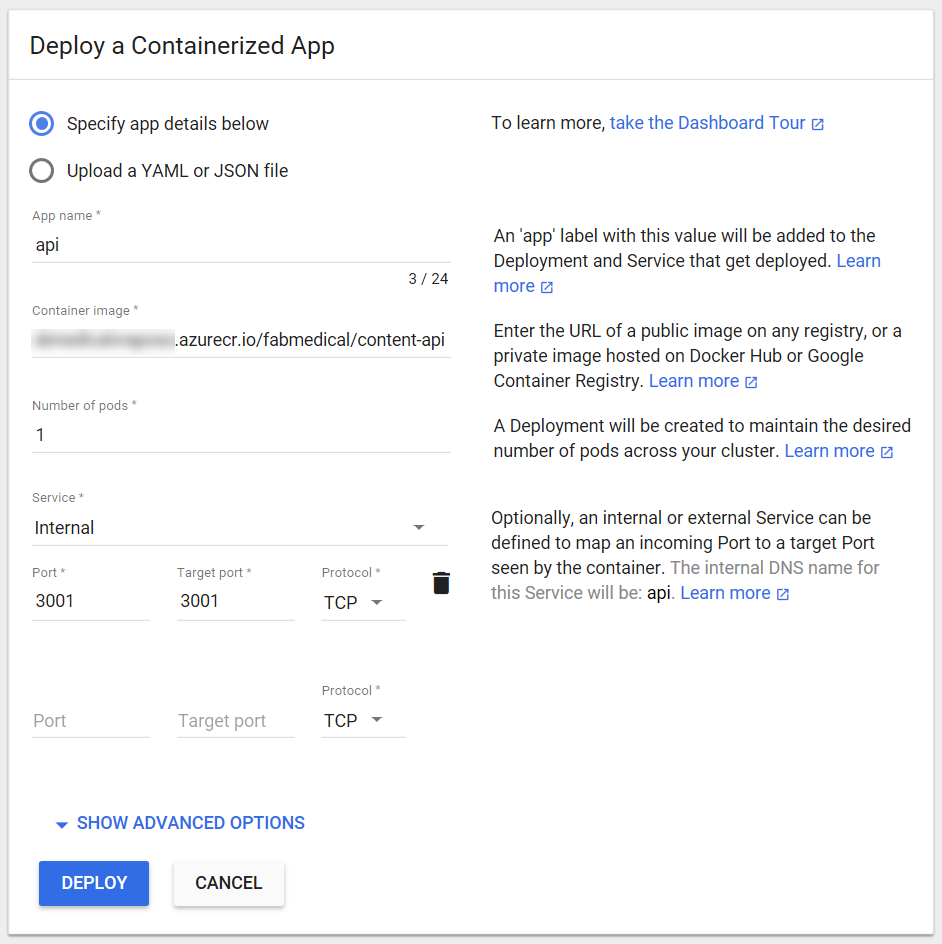


1. Open a browser window in the local machine and access the Kubernetes management dashboard at the Services view.
2. To reach the dashboard, you must access the following address:
   * 1. [http://127.0.0.1:8001/api/v1/namespaces/kube-system/services/kubernetes-dashboard/proxy/#](http://127.0.0.1:8001/api/v1/namespaces/kube-system/services/kubernetes-dashboard/proxy/)
3. If the tunnel is successful, you will see the Kubernetes management dashboard.

#### Task 2: Deploy a service using the Kubernetes management dashboard

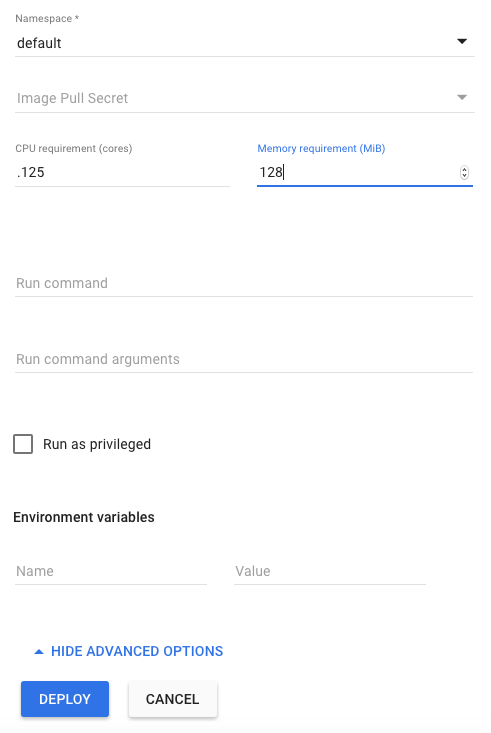
In this task, you will deploy the API application to the Azure Container Service cluster using the Kubernetes dashboard.

1. From the Kubernetes dashboard, select Create in the top right corner.

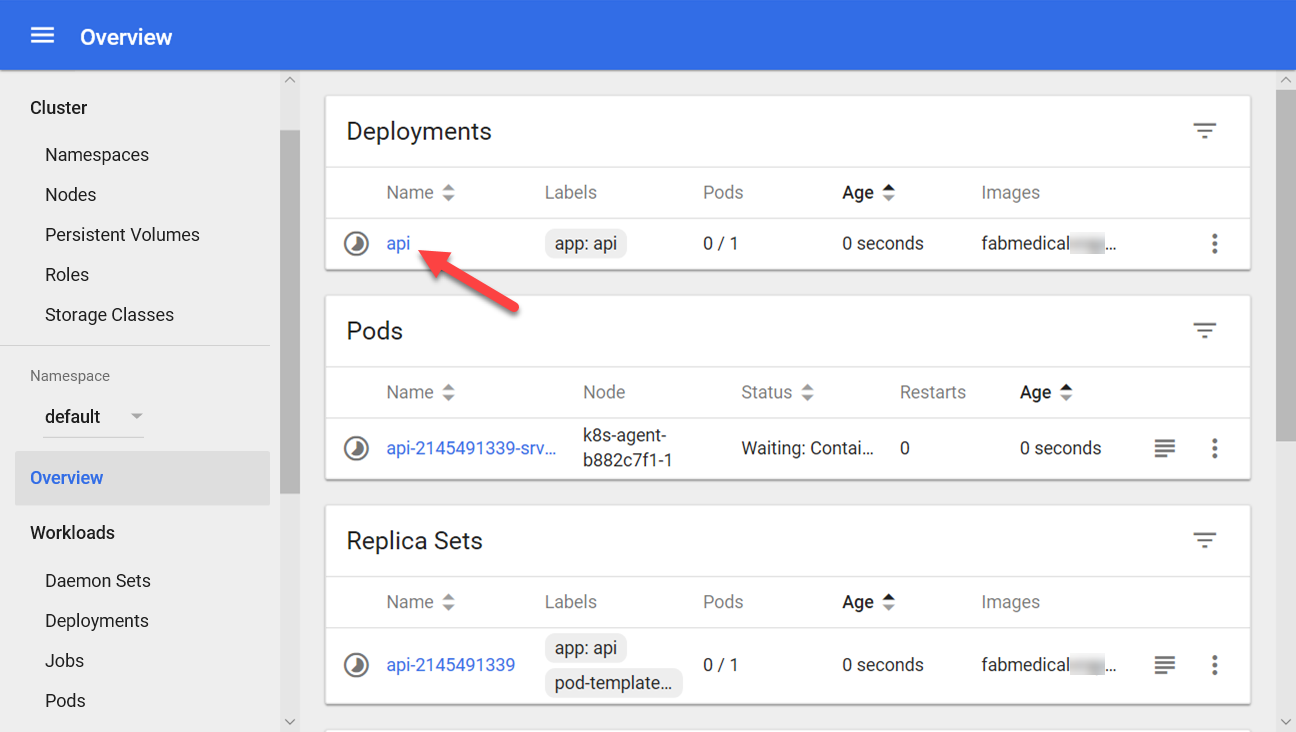


* 1. Enter “api” for the App name.
  2. Enter [LOGINSERVER]/fabmedical/content-api for the Container Image, replacing [LOGINSERVER] with your ACR login server, such as fabmedicalsol.azurecr.io
  3. Set Number of pods to 1.
  4. Set Service to “Internal”.
  5. Use 3001 for Port and 3001 for Target port.

1. Select SHOW ADVANCED OPTIONS.
   1. Enter 0.125 for the CPU requirement.
   2. Enter 128 for the Memory requirement.



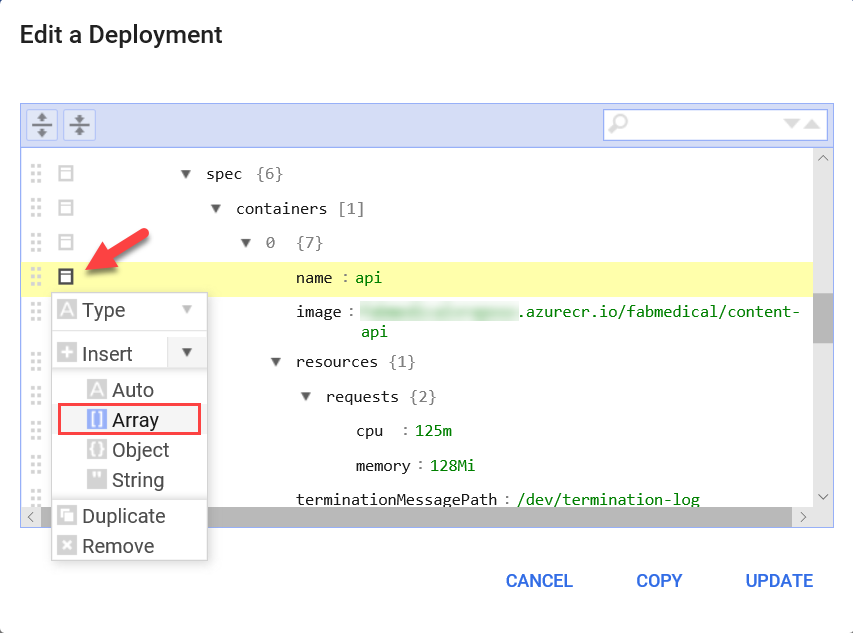
1. Select Deploy to initiate the service deployment based on the image. This can take a few minutes. In the meantime, you will be redirected to the Overview dashboard. Select the API deployment from the Overview dashboard to see the deployment in progress.



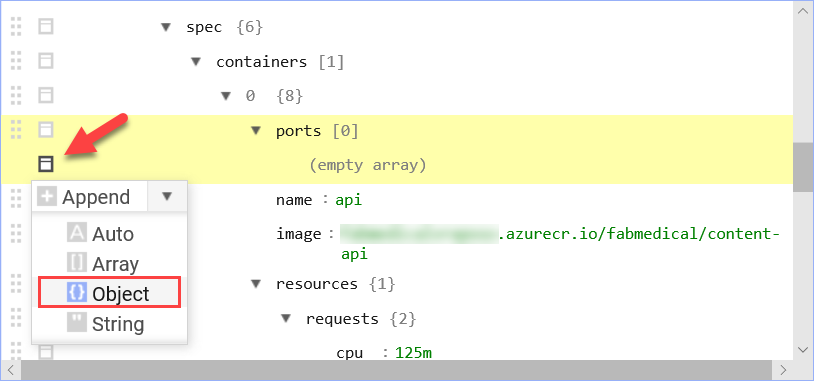
1. Configure the deployment to use a fixed host port for initial testing. Select Edit.



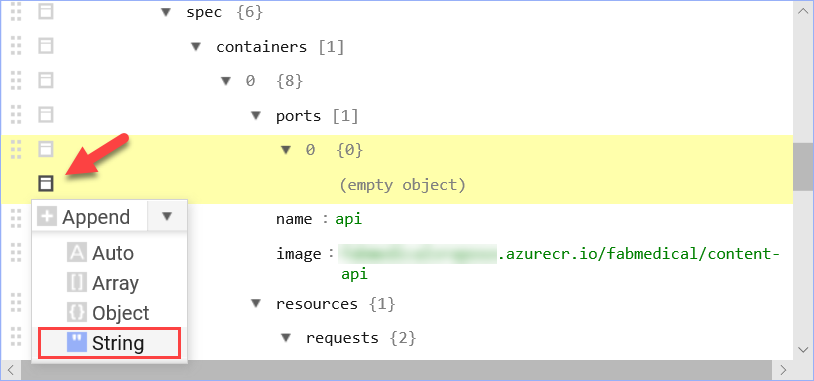
1. In the Edit a Deployment dialog you will see a list of settings shown in an expanded tree format. If you scroll down half way you will see a spec node as shown in the screenshot below. The containers spec has a single entry for the API container at the moment, shown under the first container array entry shown as 0 in the screenshot. Beneath this, you’ll see the name of the container API — this is how you know you are looking at the correct container spec.
   1. Select the menu icon on the name node and you’ll see the menu shown in the screenshot.
   2. From this menu, select the Insert dropdown.
   3. Select **Array** to insert an array inside the 0 node at the same level as the name node.



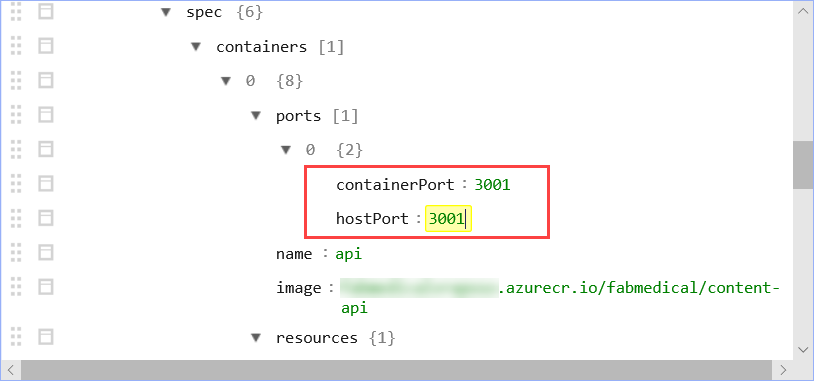
1. Name the new array “ports” and select the menu icon on the empty array row as shown in the screenshot. Select the Append dropdown and select **Object** to insert an object into the array.



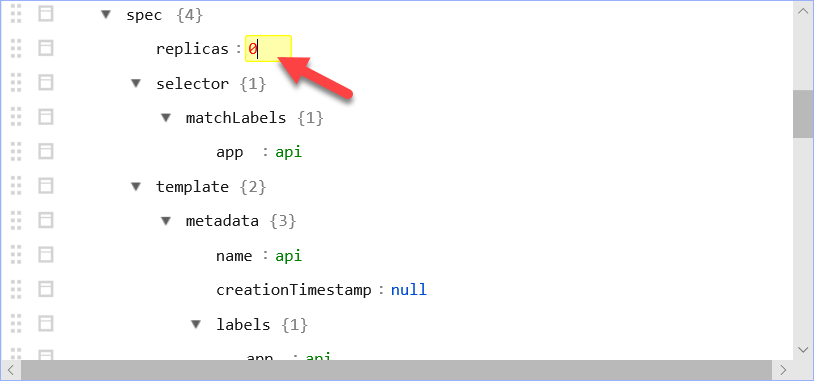
1. Use the menu to add two properties to the port array entry 0. Select the menu icon for the empty object row as shown in the screenshot. Select the Append dropdown and select String to insert a string property into the object:



1. Set the following values (repeat the append string procedure to create hostPort):
   1. **containerPort**: 3001
   2. **hostPort**:3001

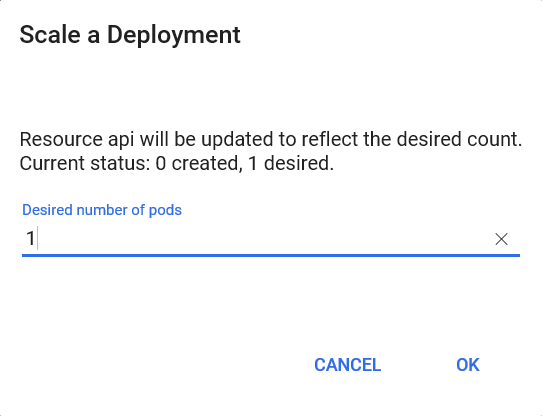


1. Set the replicas to 0 so that you can redeploy all the API pods.

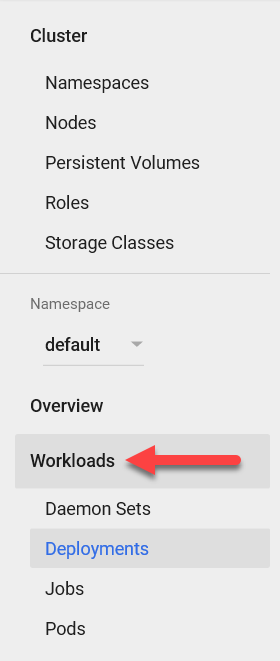


1. Select **Update** to save the changes.
2. Now, select **Scale** and, from the Scale a Deployment dialog, set the desired number of pods to 1 and select **OK**.

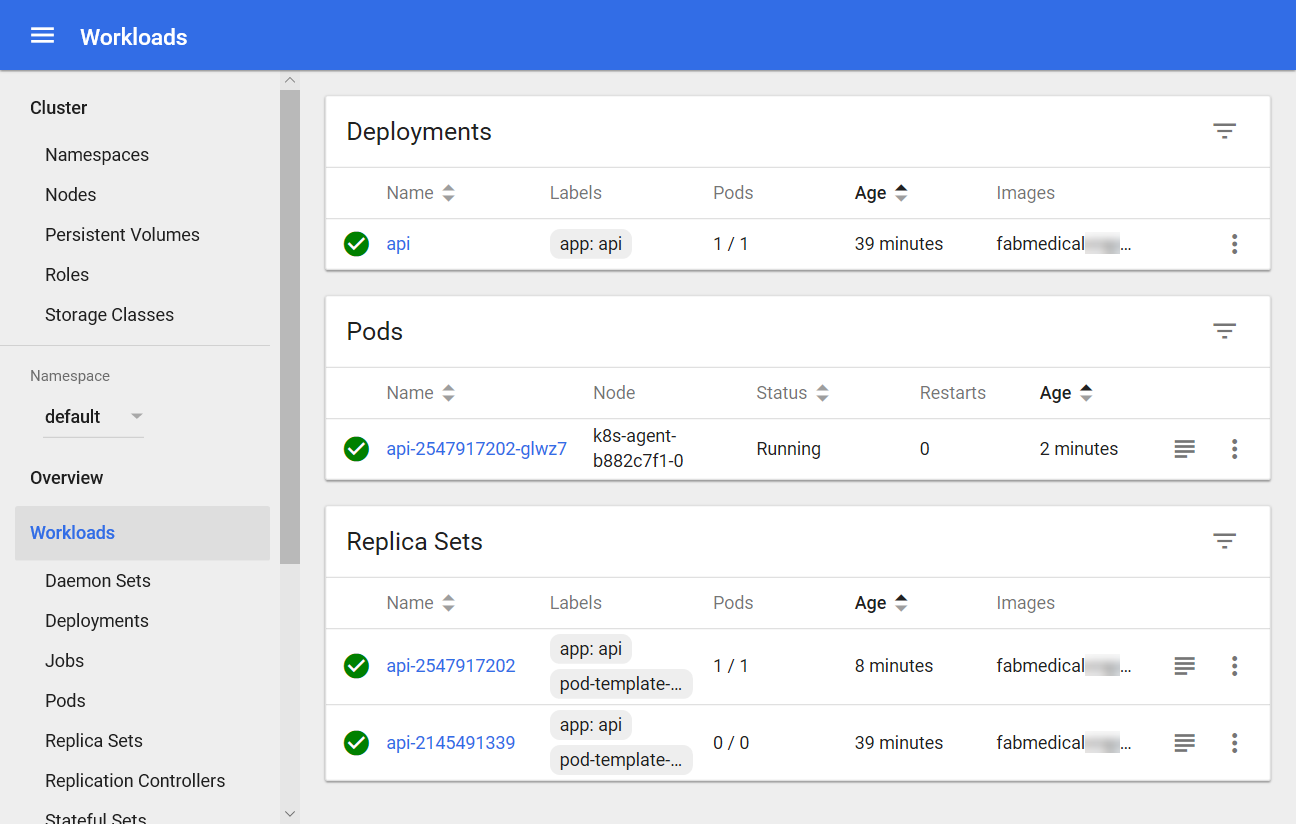




1. From the navigation menu, select the **Workloads** view.



1. From the Workloads view, you can see the deployed objects and the replica sets that correspond to your two API deployments After the deployment is successful, it will transition to a Running state as shown in the following screenshot.



#### Task 3: Deploy a service using Kubernetes REST API

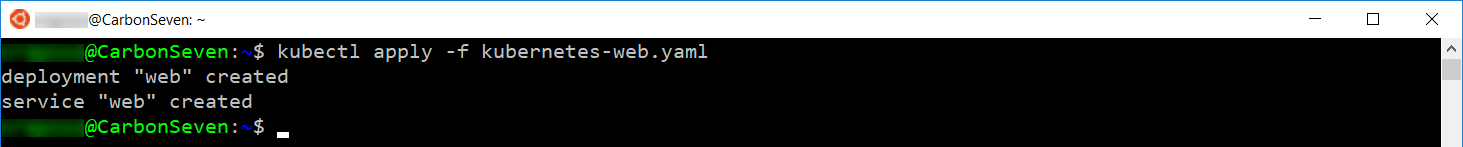
In this task, you will make HTTP requests using kubectl. These requests will deploy the web application using the REST API.

1. Open a new SSH client console and once again login to the build agent Linux VM using the Linux VM Credentials.
2. Create a text file in this directory called kubernetes-web.yaml using Vim and press the “i" key to go into edit mode.
   * 1. vi kubernetes-web.yaml
     2. <i>
3. Copy and paste the following text into the editor.

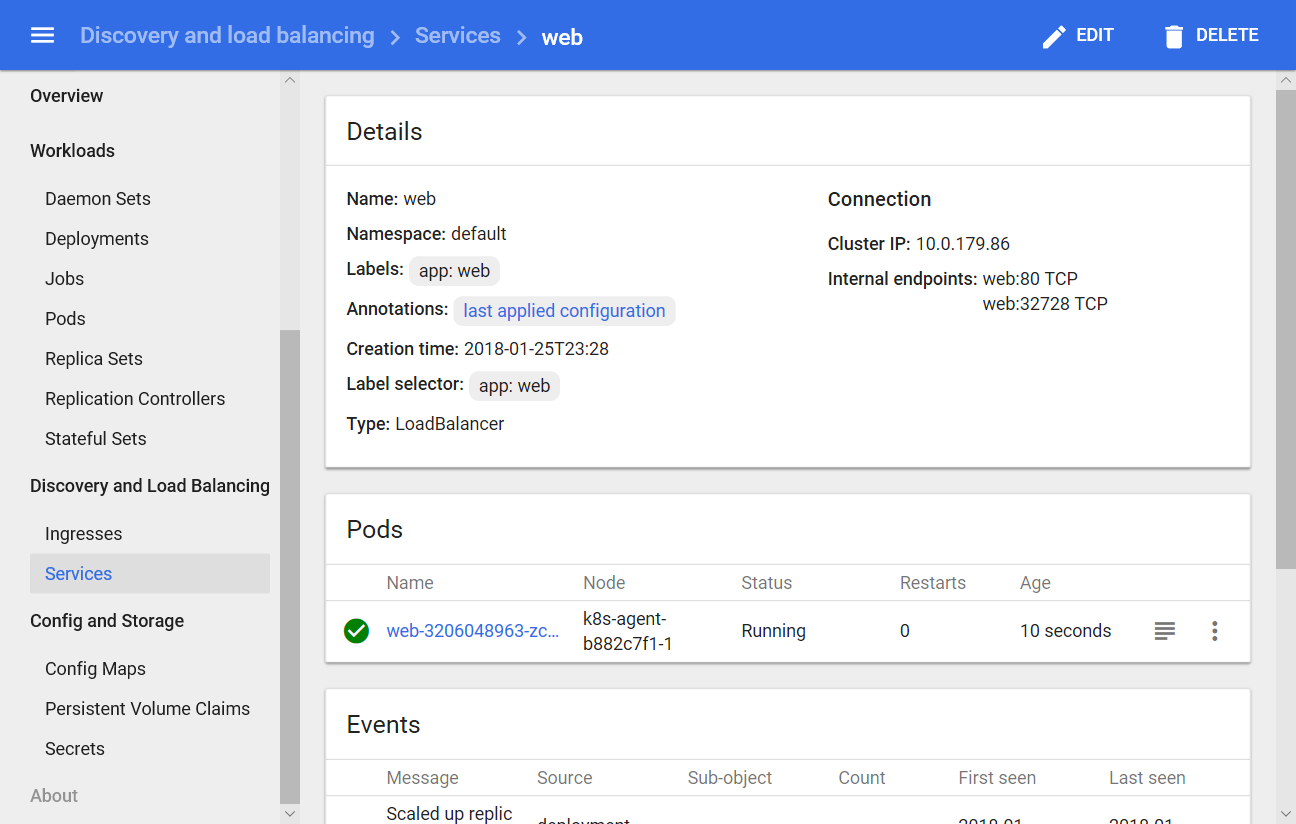
**NOTE:** be sure to copy and paste only the contents of the code block carefully to avoid introducing any special characters.

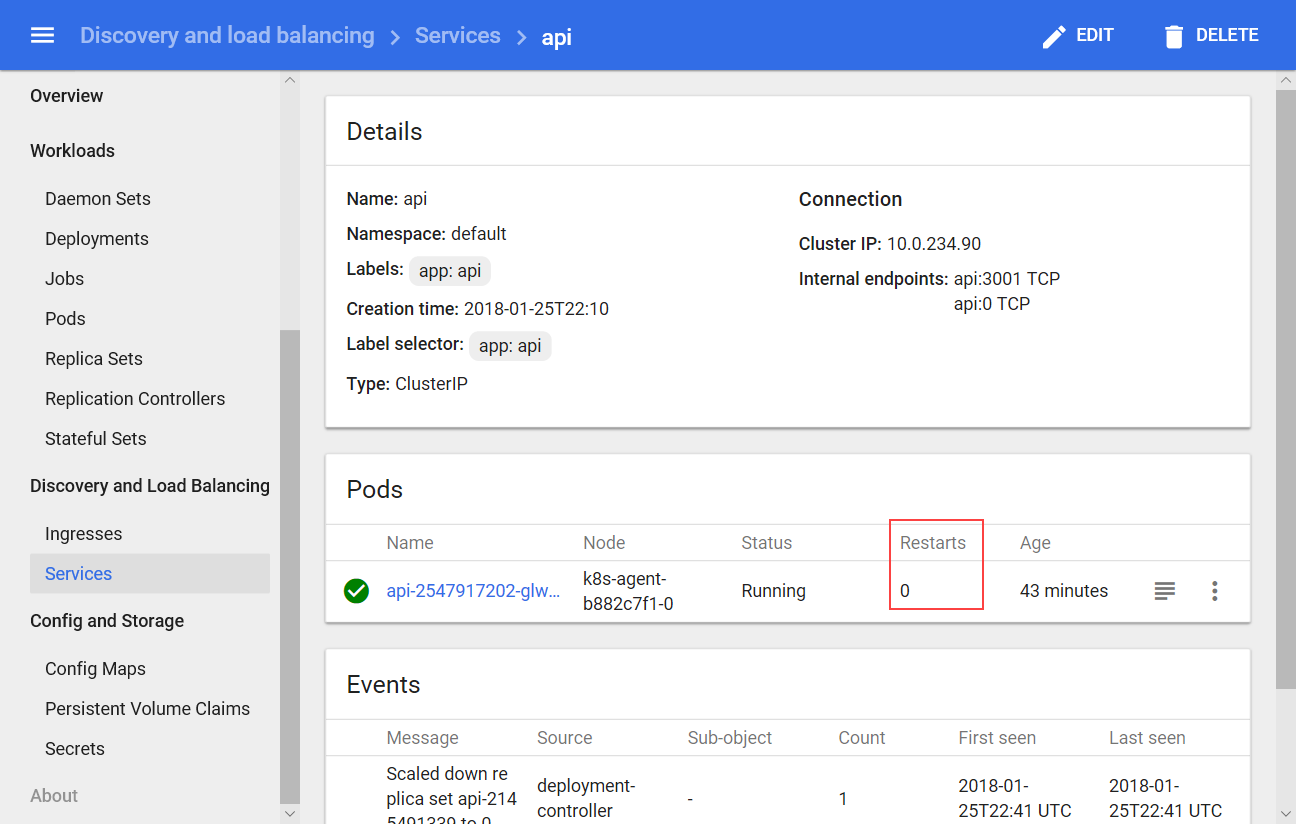
* + 1. apiVersion: extensions/v1beta1
    2. kind: Deployment
    3. metadata:
    4. labels:
    5. app: web
    6. name: web
    7. spec:
    8. replicas: 1
    9. selector:
    10. matchLabels:
    11. app: web
    12. strategy:
    13. rollingUpdate:
    14. maxSurge: 1
    15. maxUnavailable: 1
    16. type: RollingUpdate
    17. template:
    18. metadata:
    19. labels:
    20. app: web
    21. name: web
    22. spec:
    23. containers:
    24. - image: [LOGINSERVER].azurecr.io/fabmedical/content-web
    25. env:
    26. - name: CONTENT\_API\_URL
    27. value: http://api:3001
    28. livenessProbe:
    29. httpGet:
    30. path: /
    31. port: 80
    32. initialDelaySeconds: 30
    33. periodSeconds: 20
    34. timeoutSeconds: 10
    35. failureThreshold: 3
    36. imagePullPolicy: Always
    37. name: web
    38. ports:
    39. - containerPort: 80
    40. hostPort: 80
    41. protocol: TCP
    42. resources:
    43. requests:
    44. cpu: 1000m
    45. memory: 128Mi
    46. securityContext:
    47. privileged: false
    48. terminationMessagePath: /dev/termination-log
    49. terminationMessagePolicy: File
    50. dnsPolicy: ClusterFirst
    51. restartPolicy: Always
    52. schedulerName: default-scheduler
    53. securityContext: {}
    54. terminationGracePeriodSeconds: 30
    55. ---
    56. apiVersion: v1
    57. kind: Service
    58. metadata:
    59. labels:
    60. app: web
    61. name: web
    62. spec:
    63. ports:
    64. - name: web-traffic
    65. port: 80
    66. protocol: TCP
    67. targetPort: 3000
    68. selector:
    69. app: web
    70. sessionAffinity: None
    71. type: LoadBalancer

1. Edit this file and update the [LOGINSERVER] entry to match the name of your ACR login server.
2. Press the Escape key and type “:wq” and then press the Enter key to save and close the file.
   * 1. <Esc>
     2. :wq
     3. <Enter>
3. Type the following command to deploy the application described by the YAML file. You will receive a message indicating the items kubectl has created — a web deployment and a web service.
   * 1. kubectl apply -f kubernetes-web.yaml



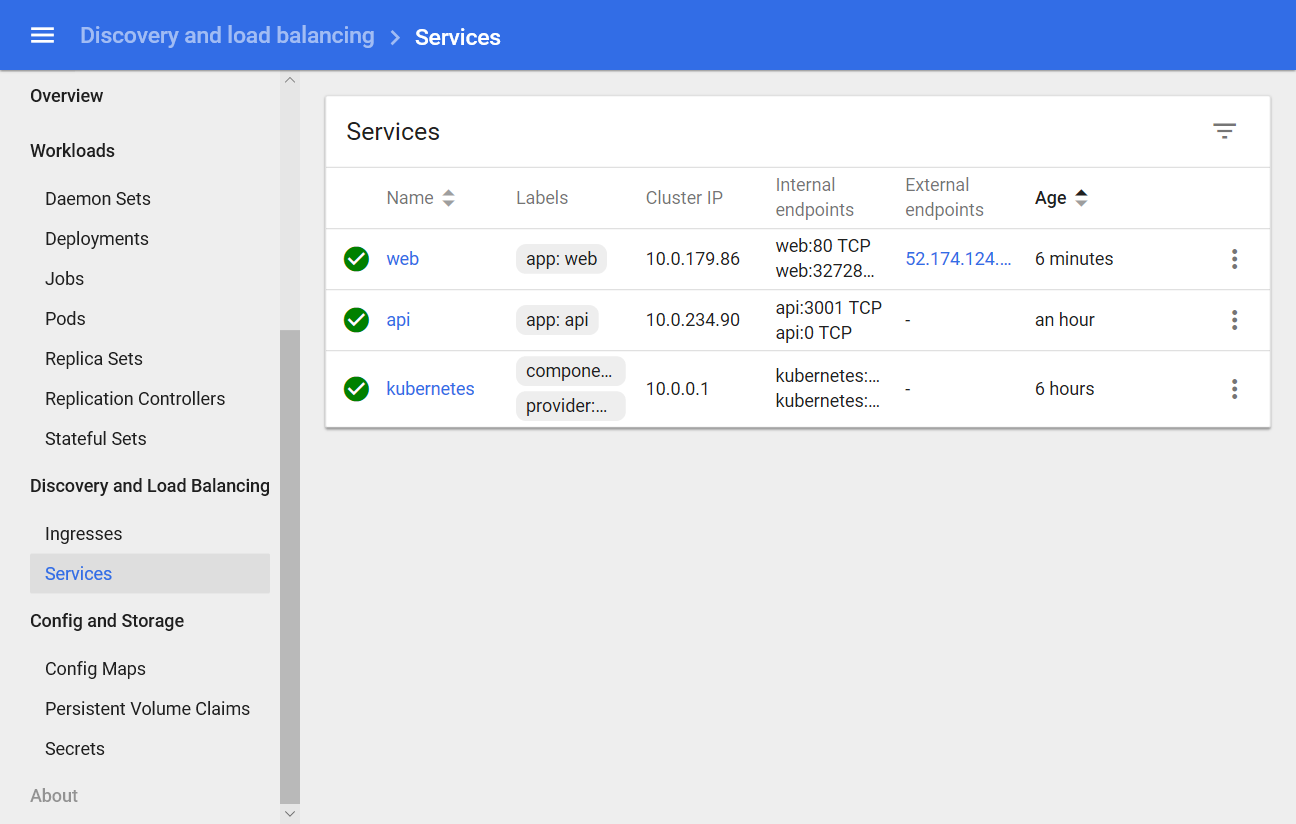
1. Return to the browser where you have the Kubernetes management dashboard open. From the navigation menu, select Services view under Discovery and Load Balancing. From the Services view, select the web service and, from this view, you will see the web service deploying. This deployment can take a few minutes. In the meantime, continue to the next step.



1. From the navigation menu, select Services view under Discovery and Load Balancing. Select the API service. While it is deploying, take note of its health status. It will be healthy as indicated by a restart count of 0. 
2. From the navigation menu, select Services view under Discovery and Load Balancing. Select the web service. At some point, if you wait long enough, you will see it is in an unhealthy state (indicated by restarts in the screenshot below). This will only happen after you fail to deploy three consecutive times, so you should not wait for this state. In the meantime, it remains in the Deploying state. You will resolve this in the next task.



1. From the navigation menu, select Services view under Discovery and Load Balancing. On the Services view, you will see two services deployed. Note that the “kubernetes” API service is always running by default to allow you to manage the cluster.

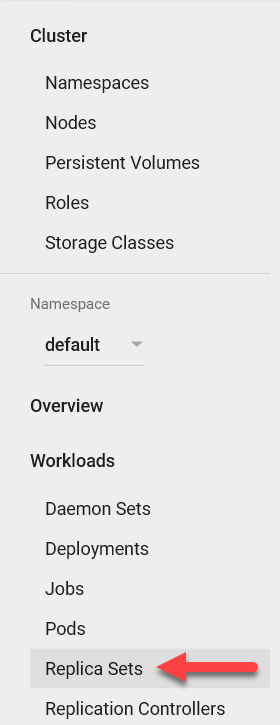


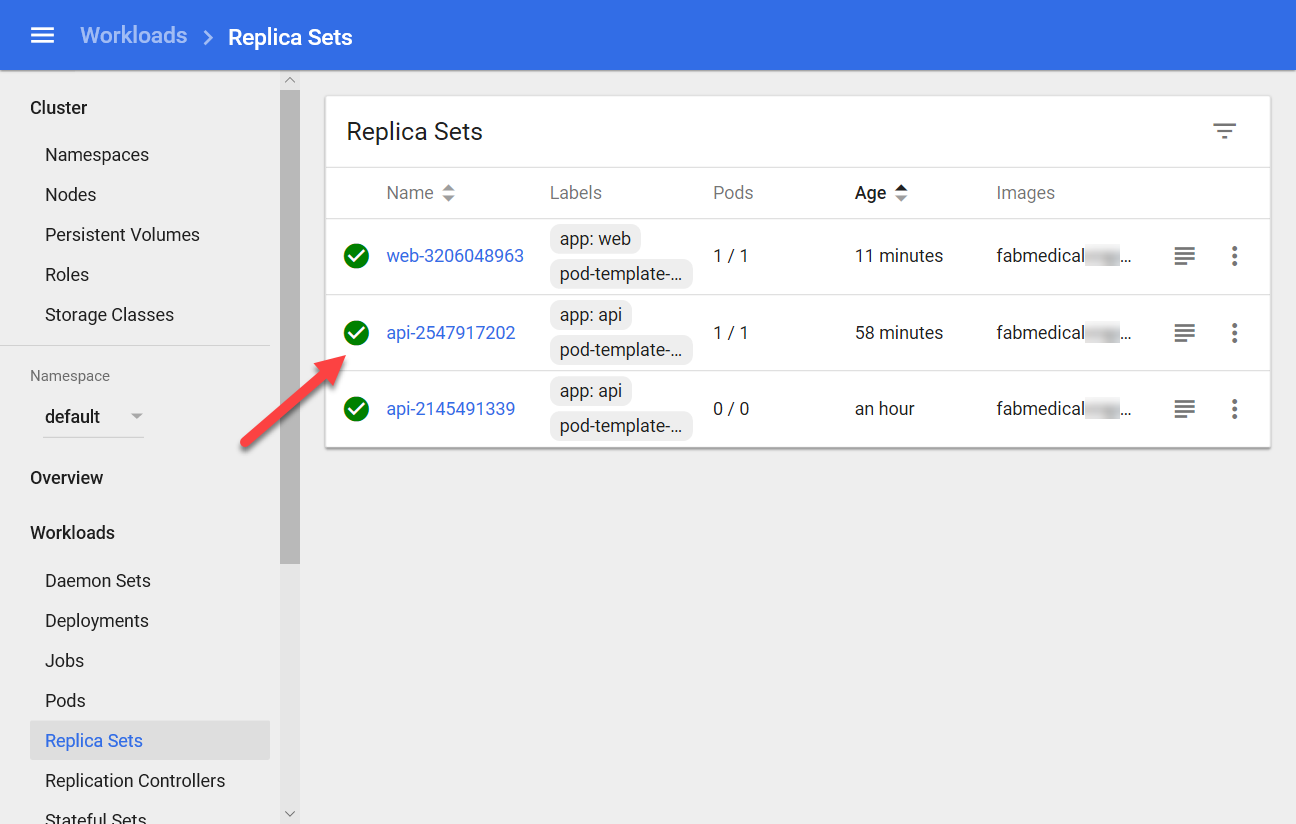
#### Task 4: Explore service instance logs and resolve an issue

In this task, you will determine why the web service deployment is unhealthy and learn how to explore logs in Kubernetes to troubleshoot issues. You will then fix the problem and redeploy the service.

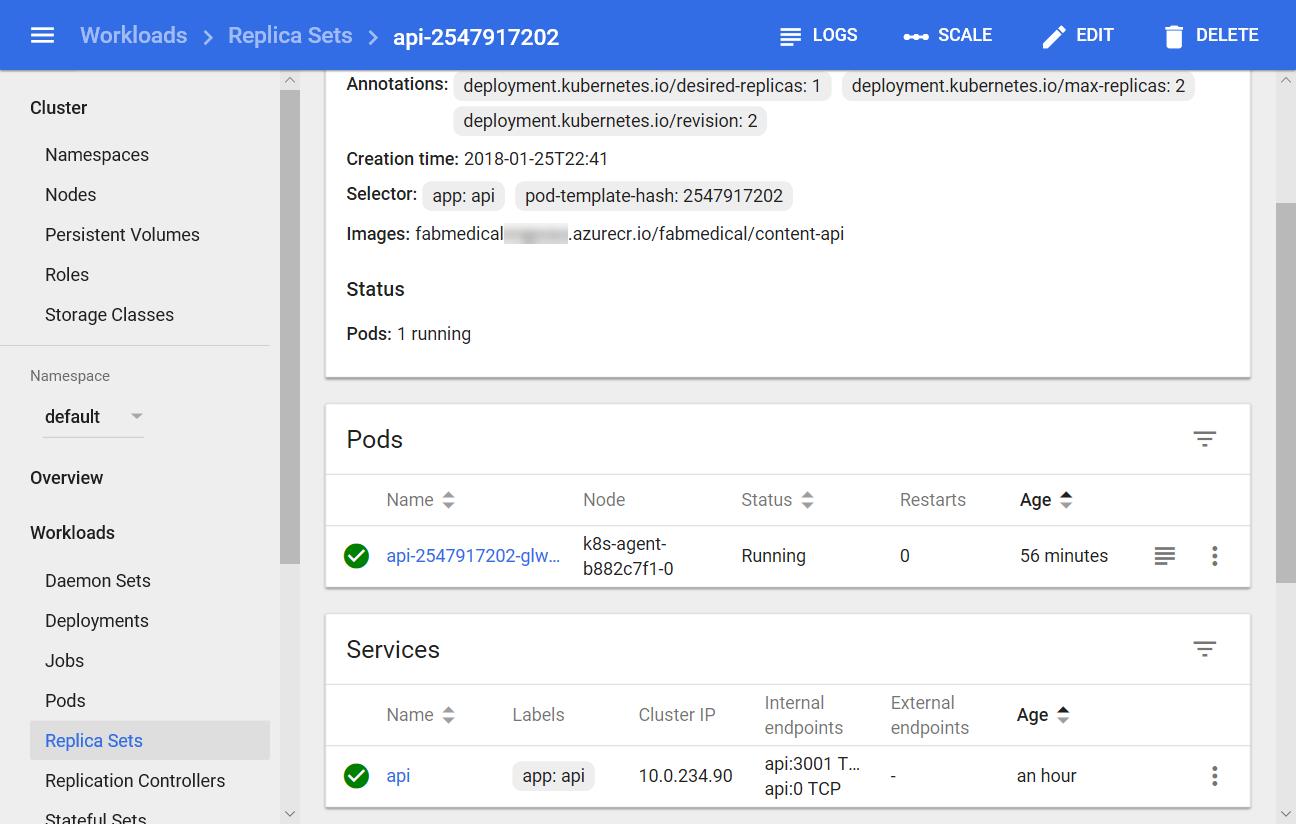
1. From the navigation menu, select the Replica Sets view under Workloads. From the Replica Sets view, select the api replica set from the list in the Replica Sets section of the view.

**NOTE:** the CPU usage and Memory usage graphs may not appear as it depends on the default configuration of the specific version of Kubernetes used by the template when you create the cluster. You can ignore if the graphs are not present.

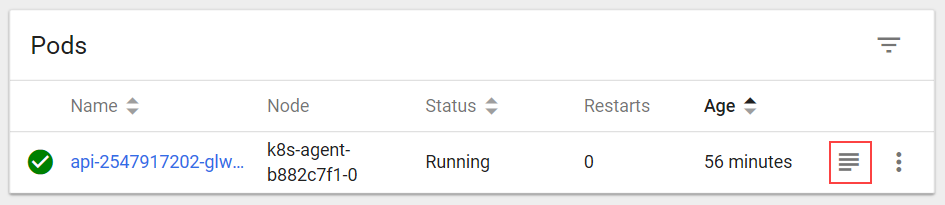




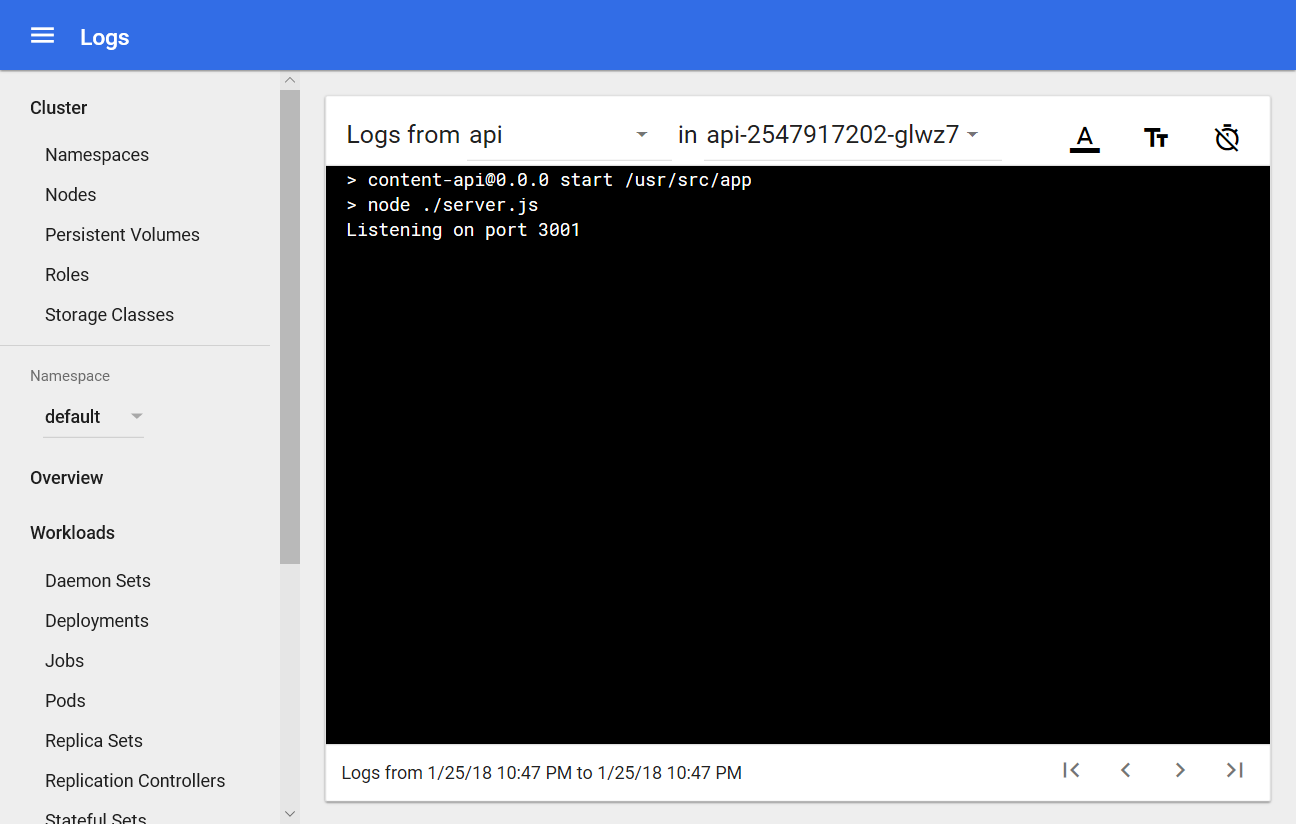
1. You will see some messages indicating that API pods were launched and are running normally.



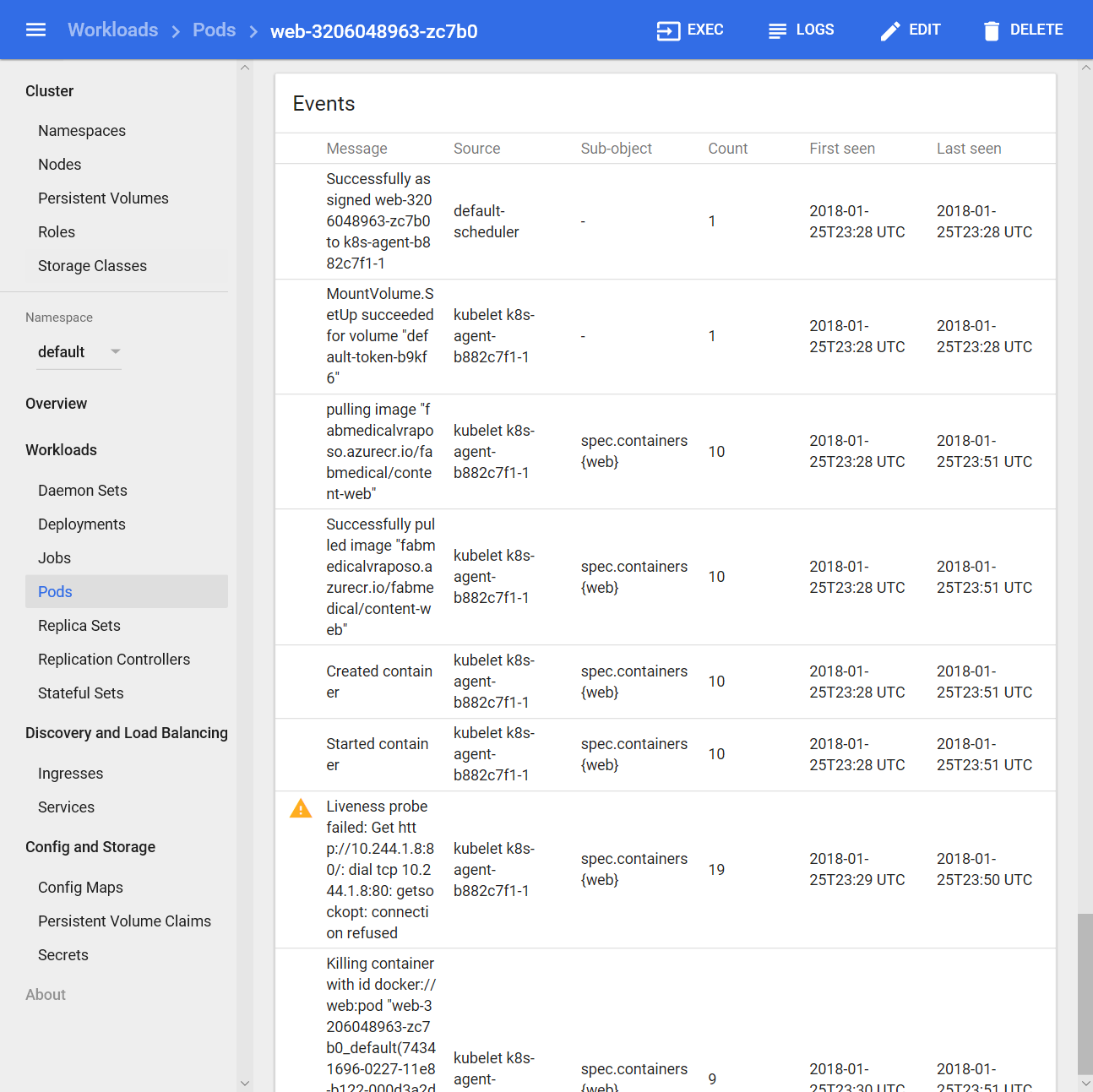
1. Select the logs icon to the right of the pod.



1. You’ll see the console output (stdout) from the initialization of the API application container. The end of the output indicates the application is listening on port 3001.



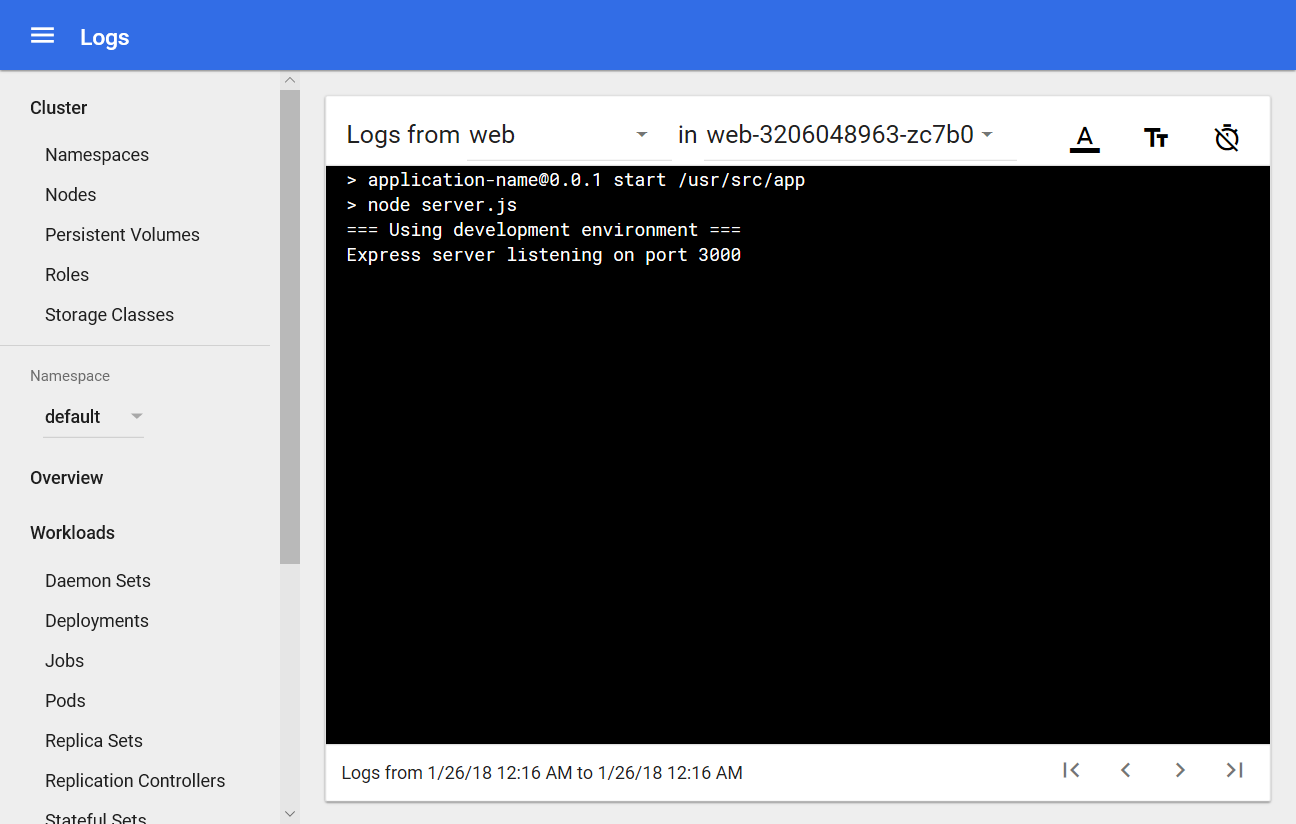
1. From the navigation menu, select Services view under Discovery and Load Balancing. Select the web service. You should see a task in an unhealthy state (indicated by the number of restarts).
2. From the navigation menu, select the Replica Sets view under Workloads. Select the web Replica Set, then select the web Pod. In the Events list, you will see warnings regarding failed health checks (indicated by orange warning triangle).



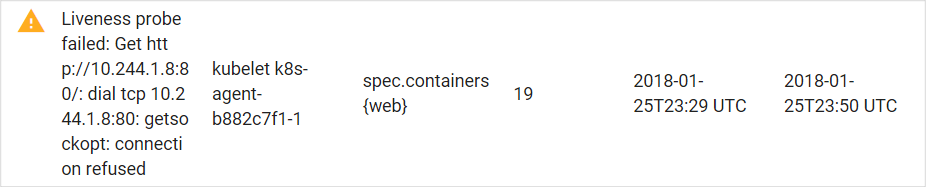
1. In the Pods section, select the logs icon for the pod to show the service’s console output.



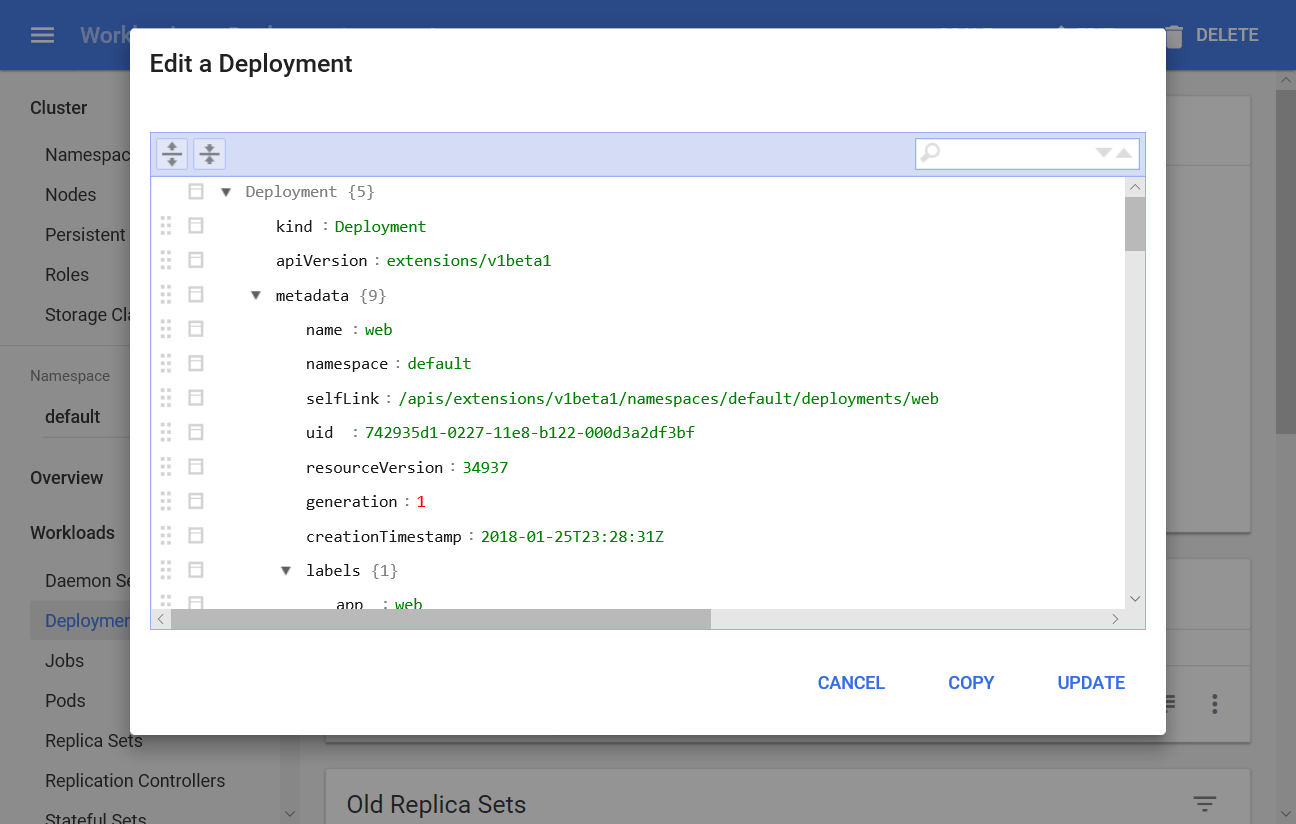
1. You’ll see a successful initialization where the service is listening at port 3000. Nothing here indicates a problem.



1. Return to the web pod view to reexamine the events shown. Note the high number of failed health check occurrences. Note that the liveness probe is querying port 80. This is not the correct port.



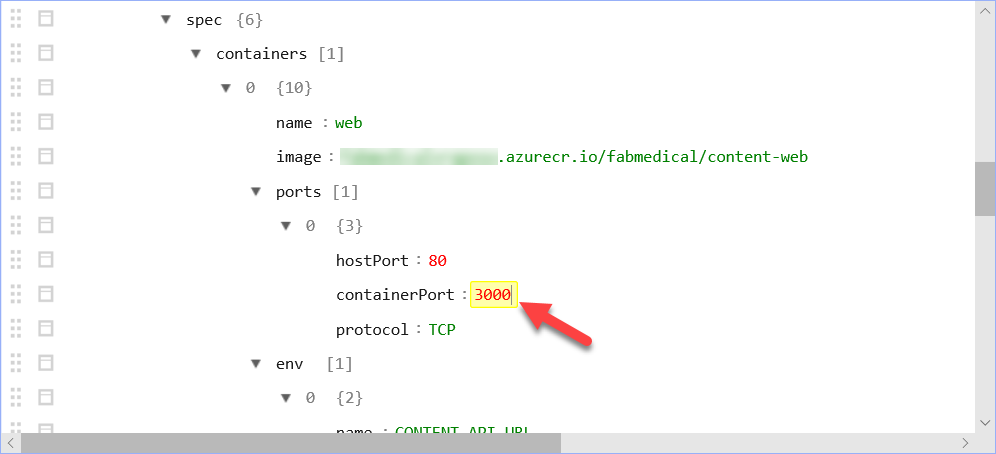
1. Return to the Deployment. Select **Edit** in the top navigation bar.



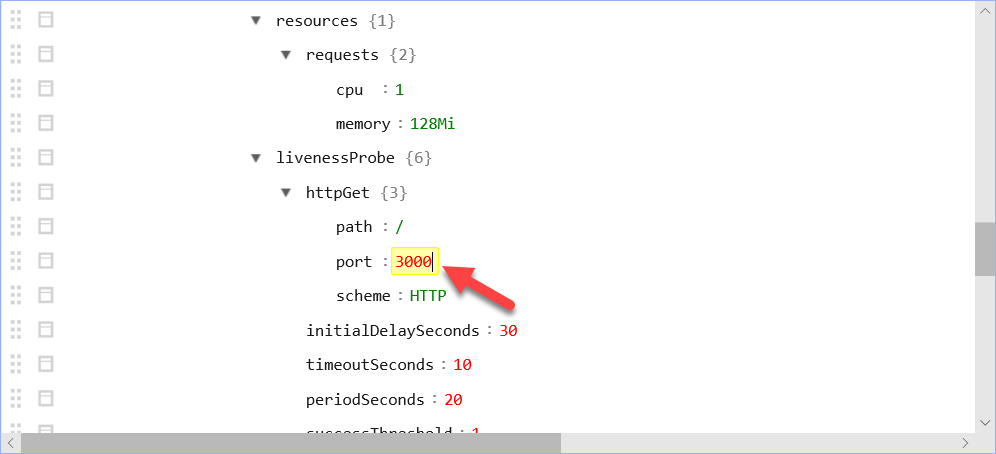
1. Scroll down to view the Port Definitions used when you published the web application. Note the container port is incorrectly set to 80, it should be 3000. Therefore, the health check is failing. The public host port is set correctly to port 80.



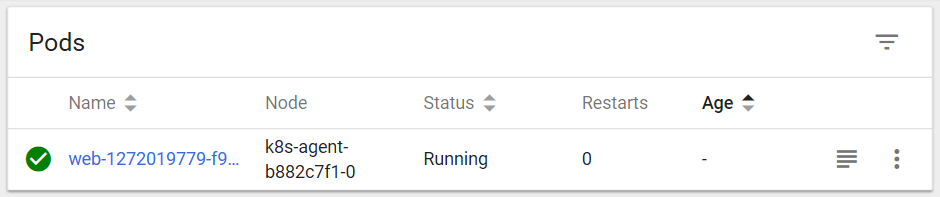
1. From this dialog, edit the ports section and set the containerPort entry to 3000.



1. In the same dialog, scroll down to the livenessProbe entry and change the port value to 3000.



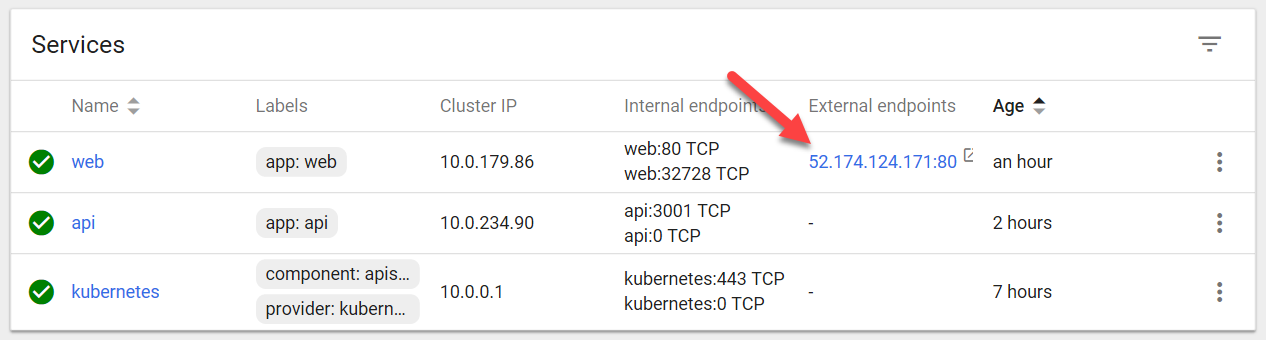
1. Select **Update** to save the changes.
2. From the navigation menu, select Workloads >Replica Sets. You’ll see that the web pod is replaced. When it is fully initialized, the pod should run normally with no restarts.
3. Select the web Replica Set. From the web Replica Set view, ensure the controller shows one pod and has 0 restarts.



#### Task 5: Test the application in a browser

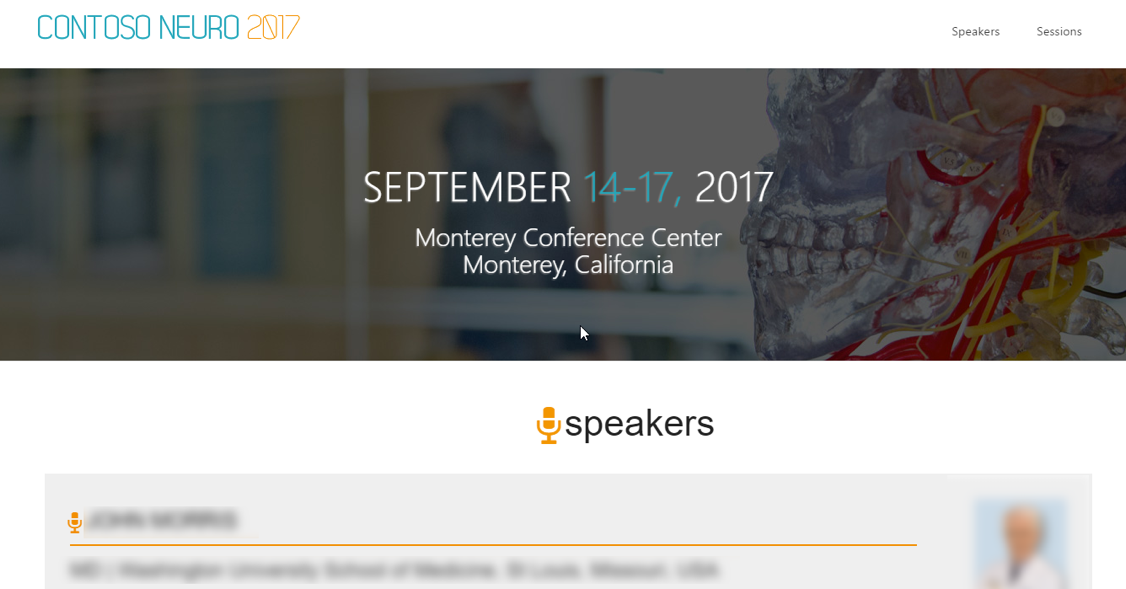
In this task, you will verify that you can browse to the web service you have deployed and view the speaker and content information exposed by the API service.

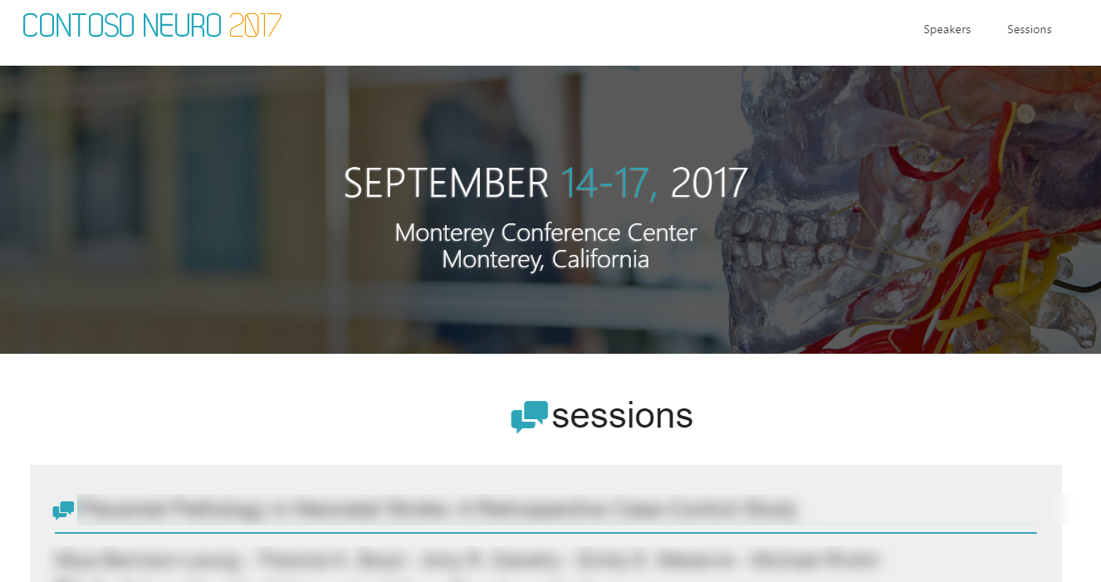
1. From the Kubernetes management dashboard, in the navigation menu, select the Services view under Discovery and Load Balancing.
2. In the list of services, locate the external endpoint for the web service and select this hyperlink to launch the application.



1. You will see the web application in your browser and be able to select the Speakers and Sessions links to view those pages without errors. The lack of errors means that the web application is correctly calling the API service to show the details on each of those pages.







## Exercise 4: Scale the application and test HA

**Duration**: 20 minutes

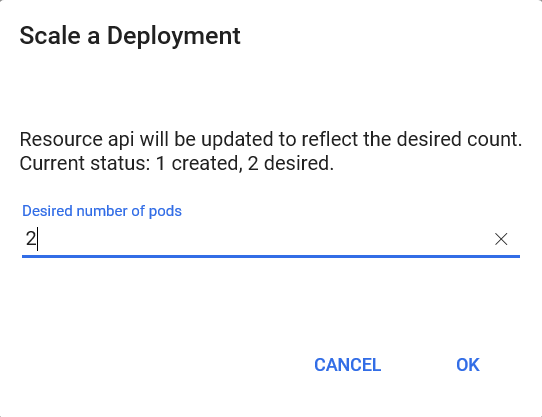
At this point you have deployed a single instance of the web and API service containers. In this exercise, you will increase the number of container instances for the web service and scale the front end on the existing cluster.

#### Task 1: Increase service instances from the Kubernetes dashboard

In this task, you will increase the number of instances for the API deployment in the Kubernetes management dashboard. While it is deploying, you will observe the changing status.

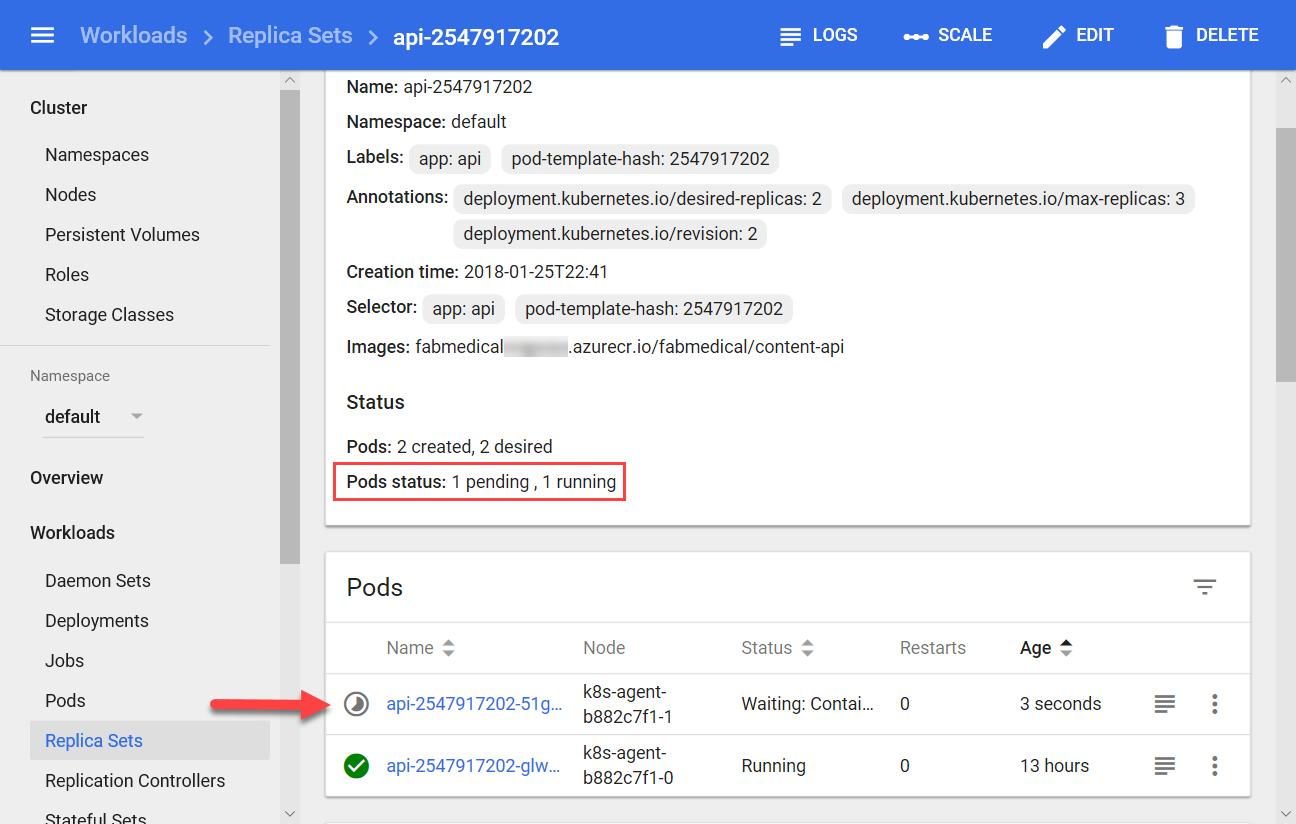
1. From the navigation menu, select Workloads>Deployments and select the API deployment.
2. Select SCALE.



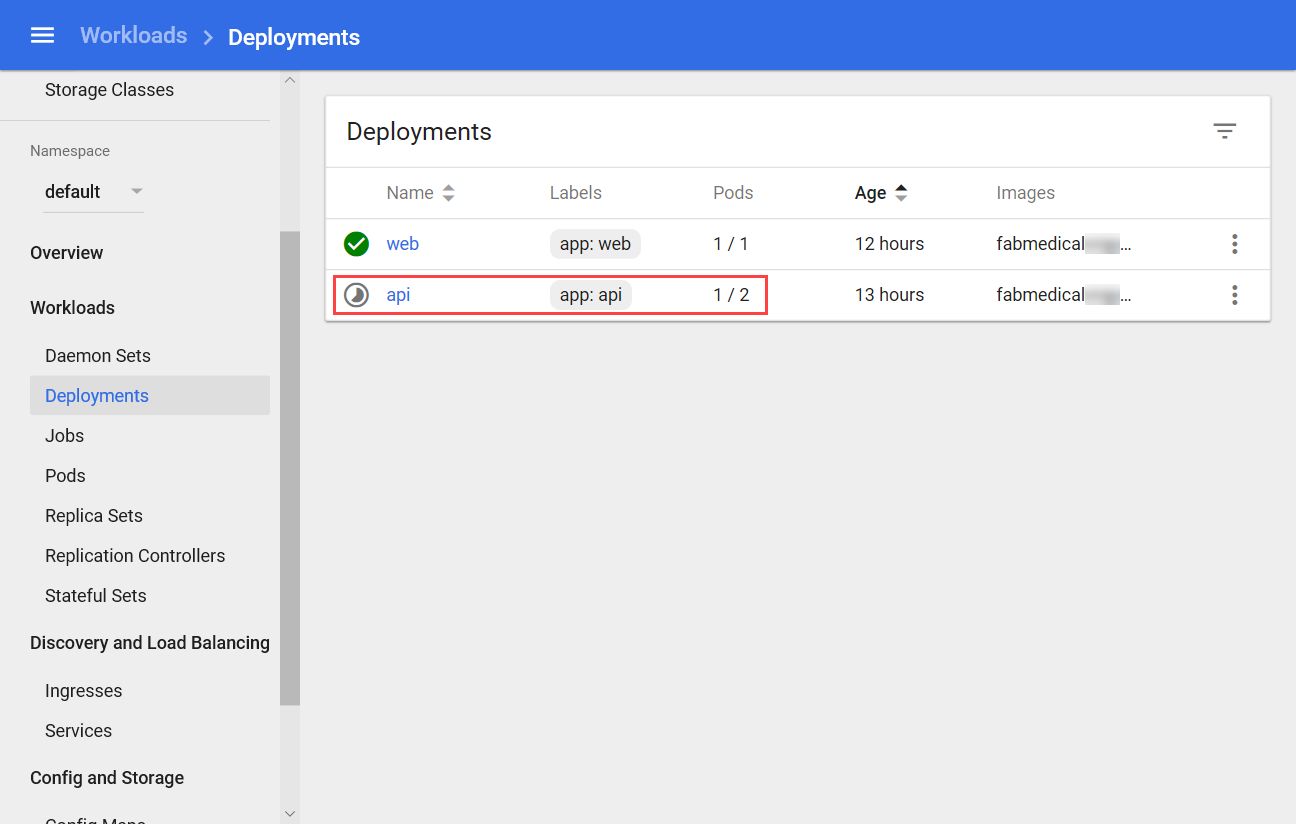
1. Change the number of pods to 2, and then select **OK**. 

**NOTE: If the deployment completes quickly, you may not see the deployment Waiting states in the dashboard as described in the following steps**.

1. From the Replica Set view for the API, you’ll see it is now deploying and that there is one healthy instance and one pending instance.



1. From the navigation menu, select Deployments and from the list and note that the api service has a pending status indicated by the grey timer icon, and it shows a pod count 1 of 2 instances (shown as “1/2”).



1. From the Navigation menu, select Workloads. From this view note that the health overview in the right panel of this view. You’ll see the following:
   1. One deployment and one replica set are each healthy for the api service.
   2. One replica set is healthy for the web service.
   3. Three pods are healthy.
2. Navigate to the web application from the browser again. The application should still work without errors as you navigate to Speakers and Sessions pages.
   1. Navigate to the /stats.html page. You’ll see information about the environment including:
      1. webTaskId: the task identifier for the web service instance
      2. taskId: the task identifier for the API service instance
      3. hostName: the hostname identifier for the API service instance
      4. pid: the process id for the API service instance
      5. mem: some memory indicators returned from the API service instance
      6. counters: counters for the service itself, as returned by the API service instance
      7. uptime: the up time for the API service
   2. Refresh the page in the browser, and you can see the hostName change between the two API service instances. The letters after “api-{number}-” in the hostname will change.

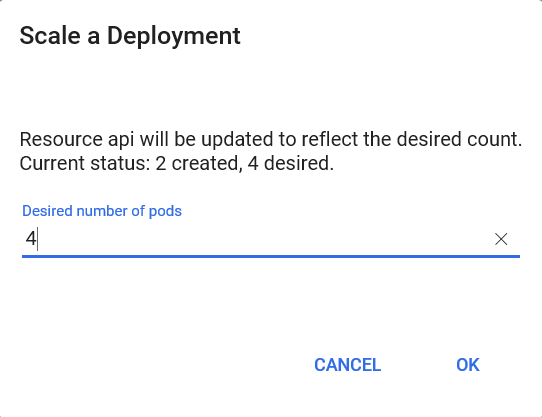
#### Task 2: Increase service instances beyond available resources

In this task, you will try to increase the number of instances for the API service container beyond available resources in the cluster. You will observe how Kubernetes handles this condition and correct the problem.

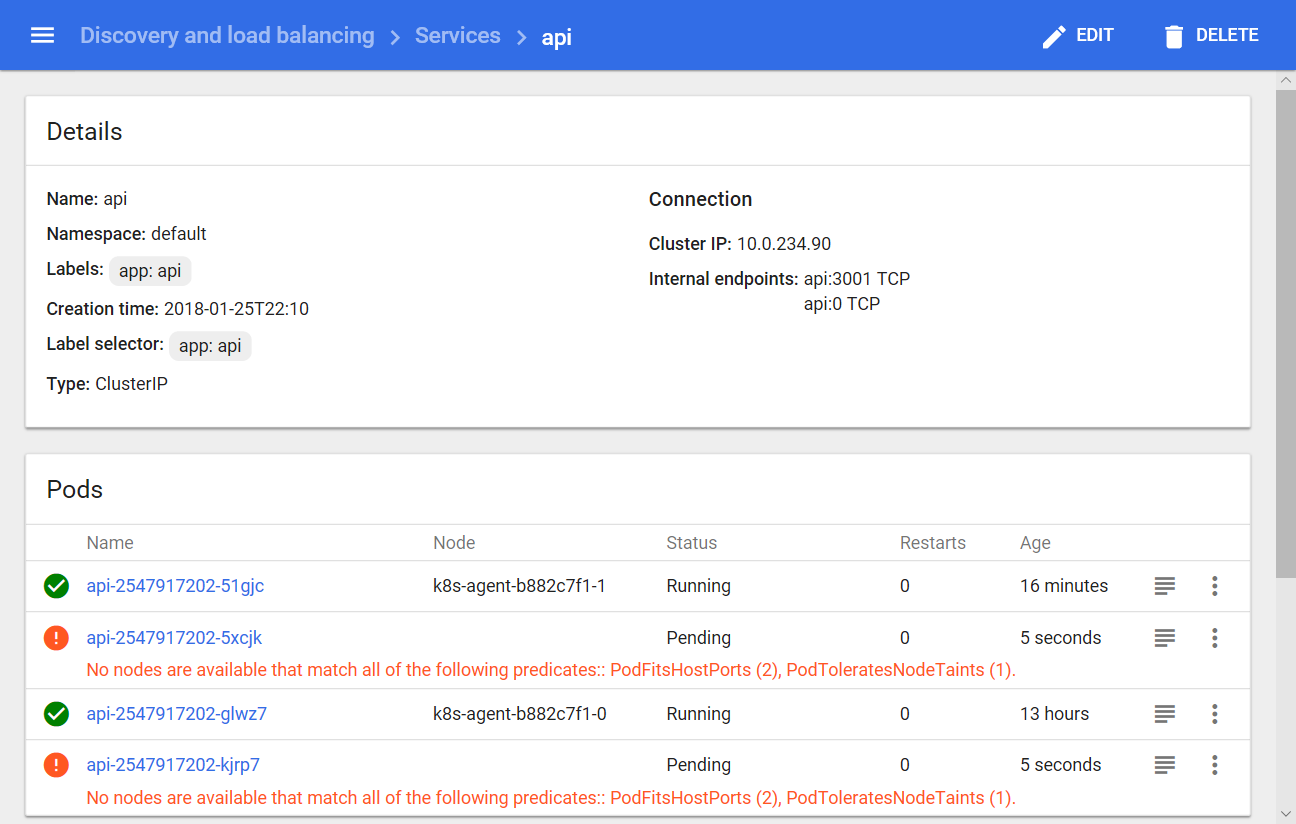
1. From the navigation menu, select Deployments. From this view, select the API deployment.
2. From the API deployment view, select **Scale**.



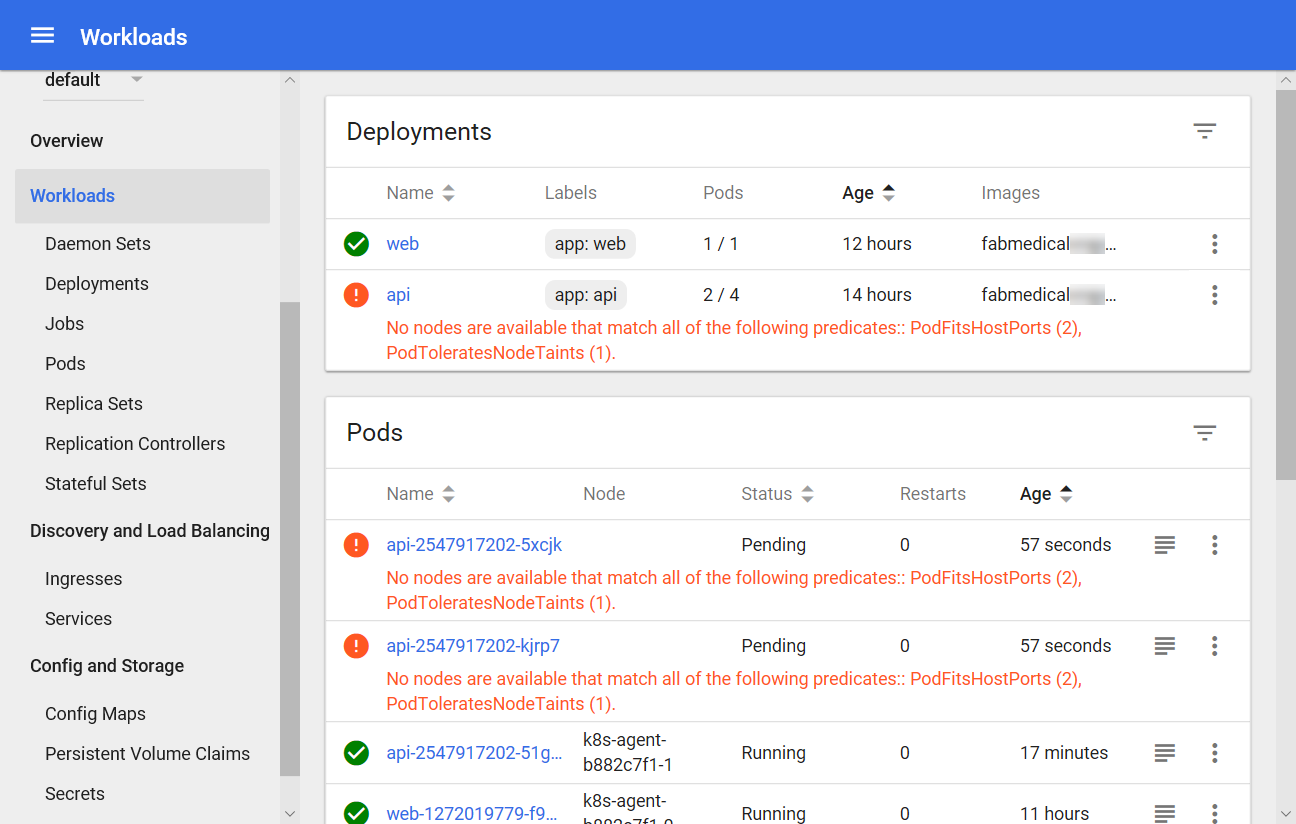
1. Change the number of pods to 4 and select **OK**.



1. From the navigation menu, select Services view under Discovery and Load Balancing. Select the api service from the Services list. From the api service view, you’ll see it has two healthy instances and two unhealthy (or possibly pending depending on timing) instances.

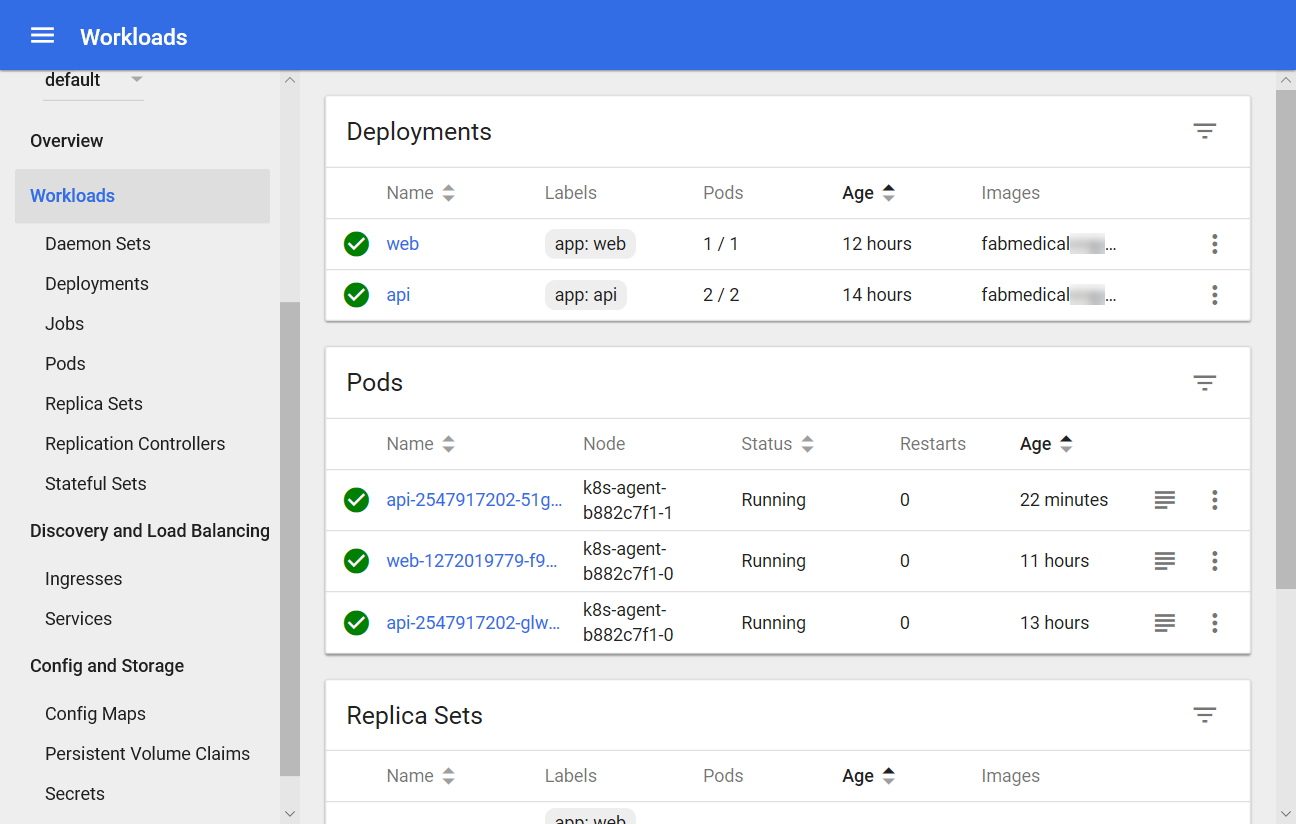


1. After a few minutes, select Workloads from the navigation menu. From this view, you should see an alert reported for the api deployment.



**NOTE: This message indicates that there weren’t enough available resources to match the requirements for a new pod instance. In this case, this is because the instance requires port 3001, and since there are only 2 nodes available in the cluster, only two api instances can be scheduled. The third and fourth pod instances will wait for a new node to be available that can run another instance using that port.**

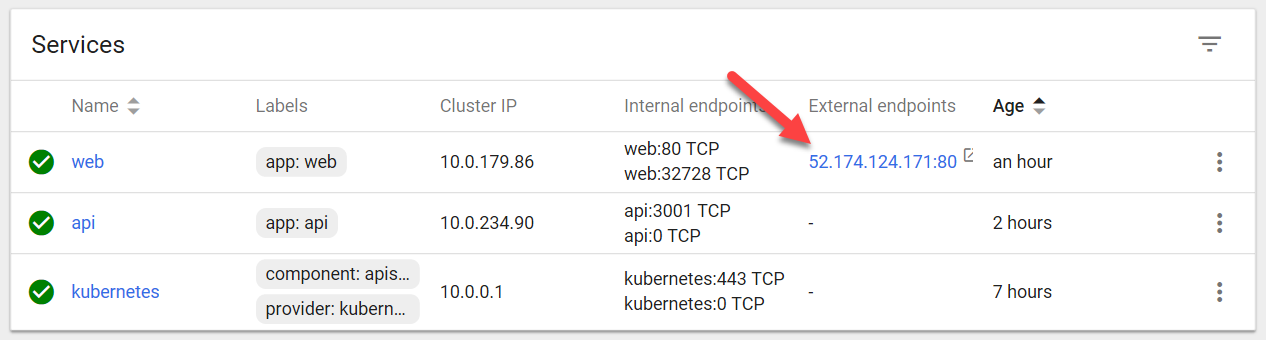
1. Reduce the number of requested pods to 2 using the Scale button.
2. Almost immediately, the warning message from the Workloads dashboard should disappear, and the API deployment will show 2/2 pods are running.

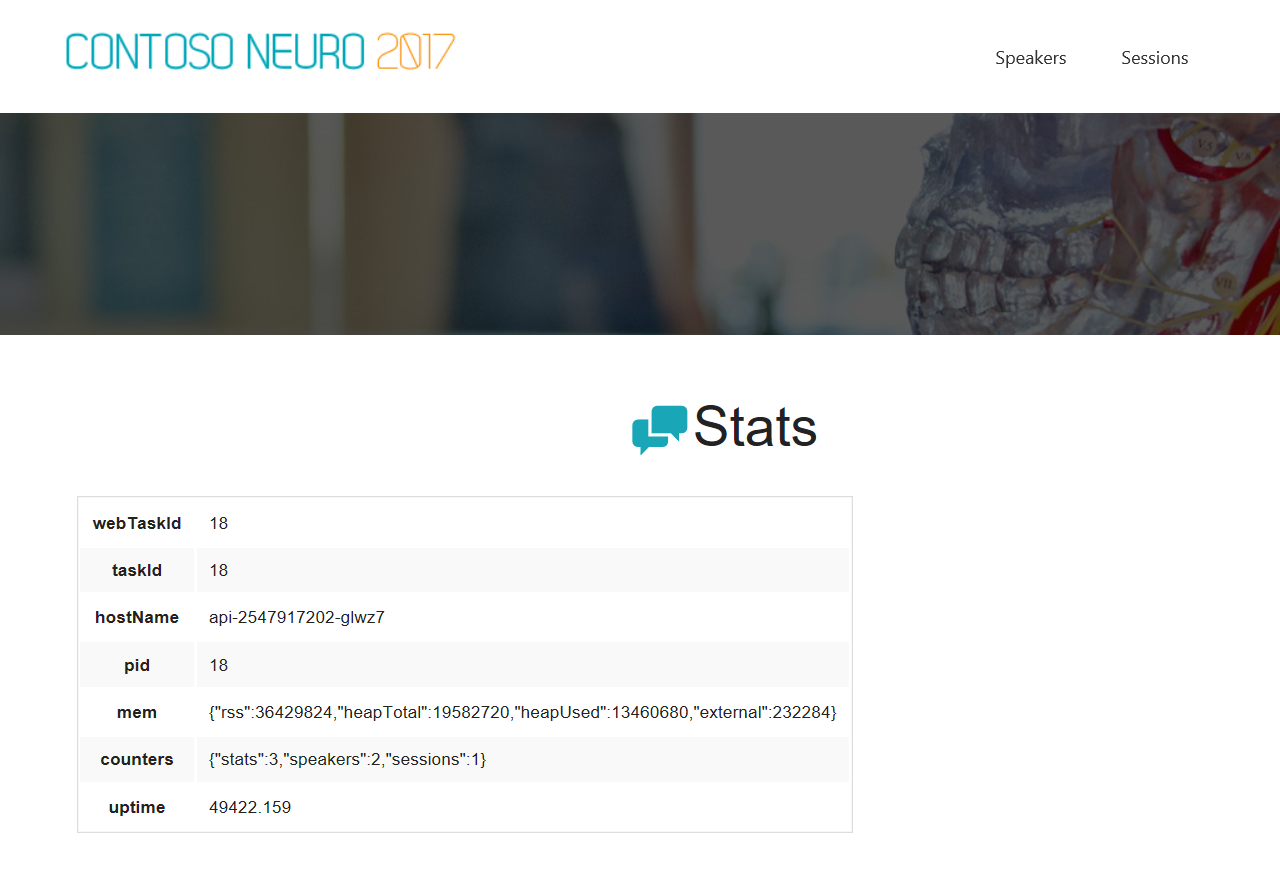


#### Task 3: Restart containers and test HA

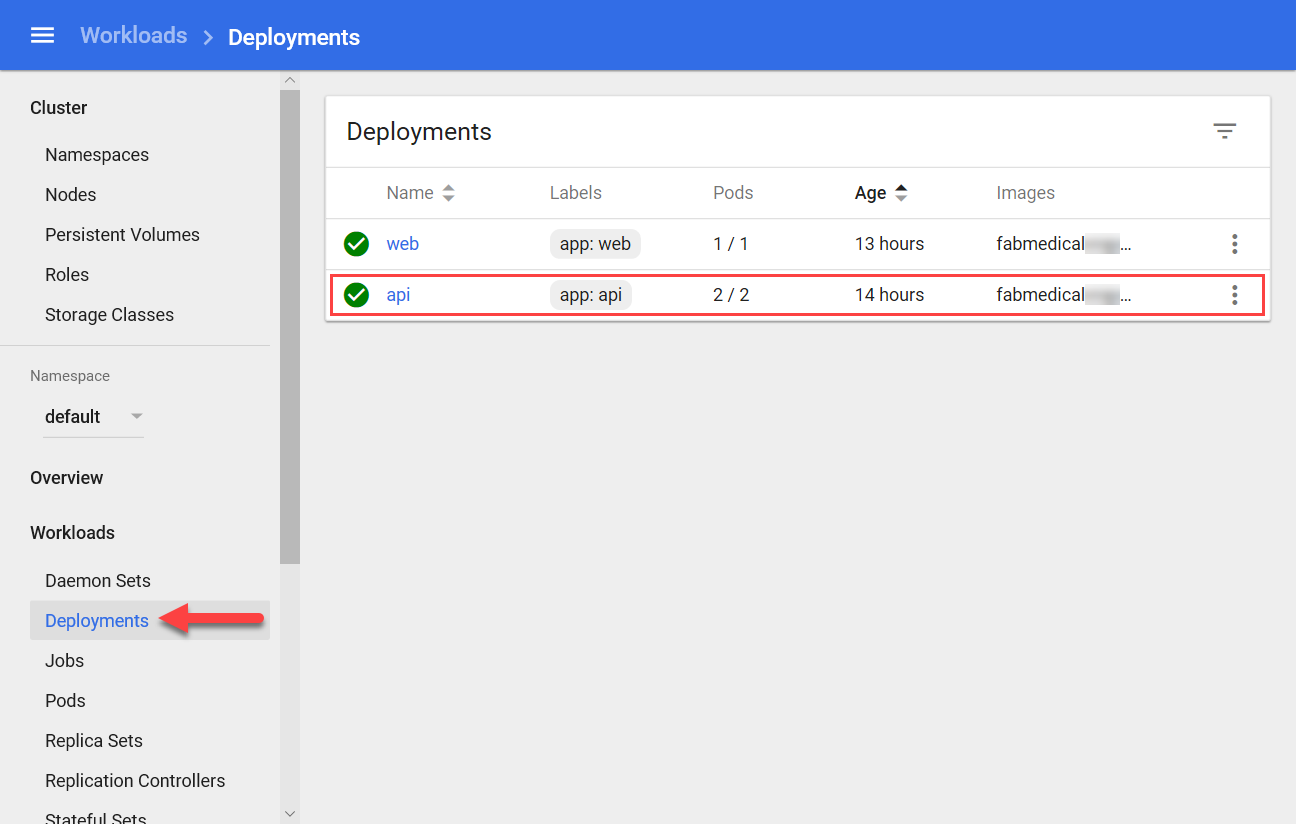
In this task, you will restart containers and validate that the restart does not impact the running service.

1. From the navigation menu on the left, select Services view under Discovery and Load Balancing. From the Services list, select the external endpoint hyperlink for the web service and visit the stats page by adding /stats.html to the URL. Keep this open and handy to be refreshed as you complete the steps that follow.

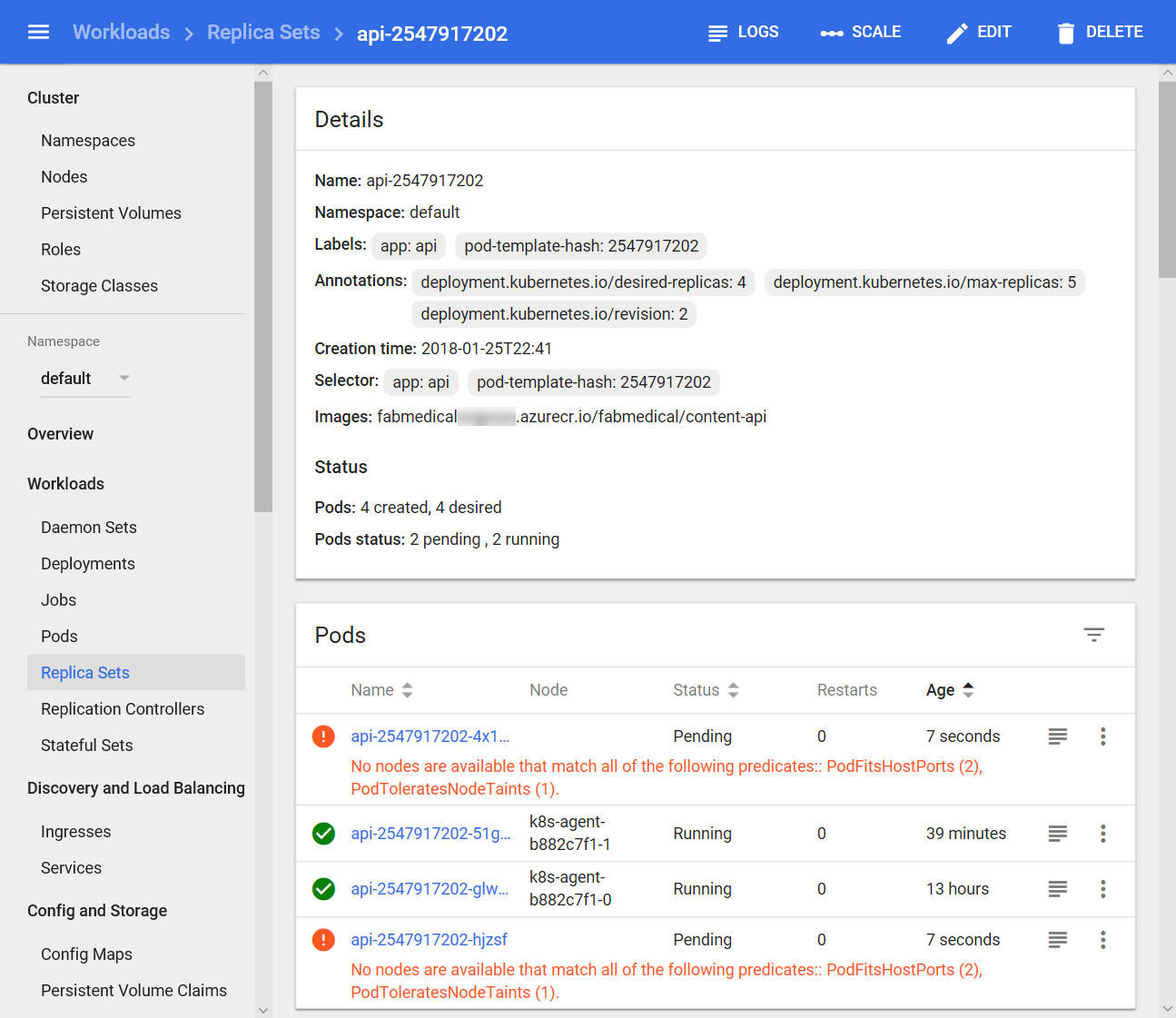




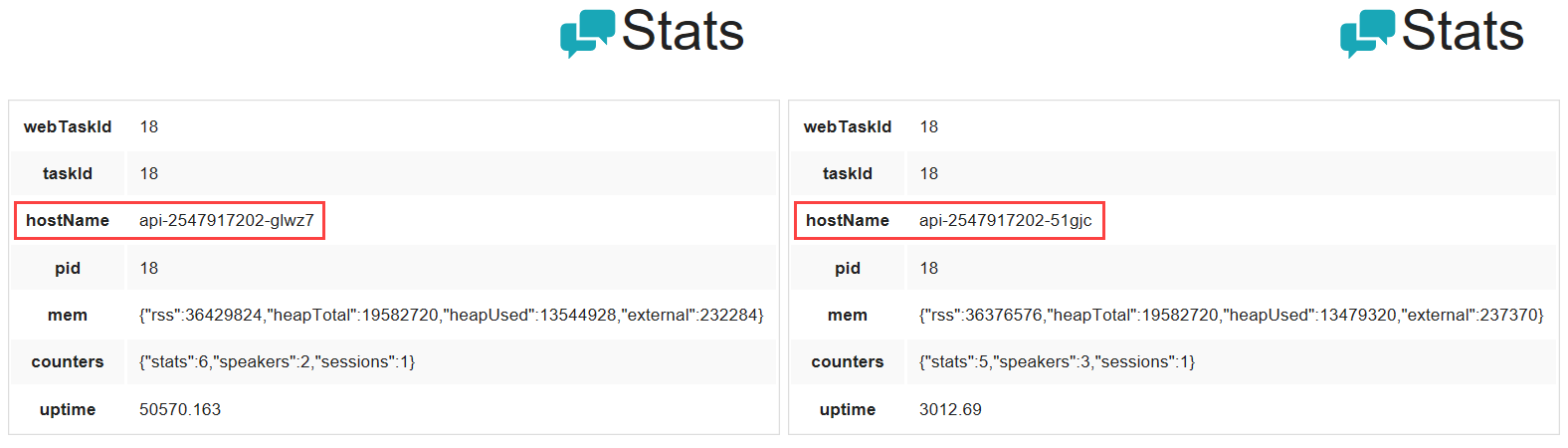
1. From the navigation menu, select Workloads>Deployments. From Deployments list, select the API deployment.



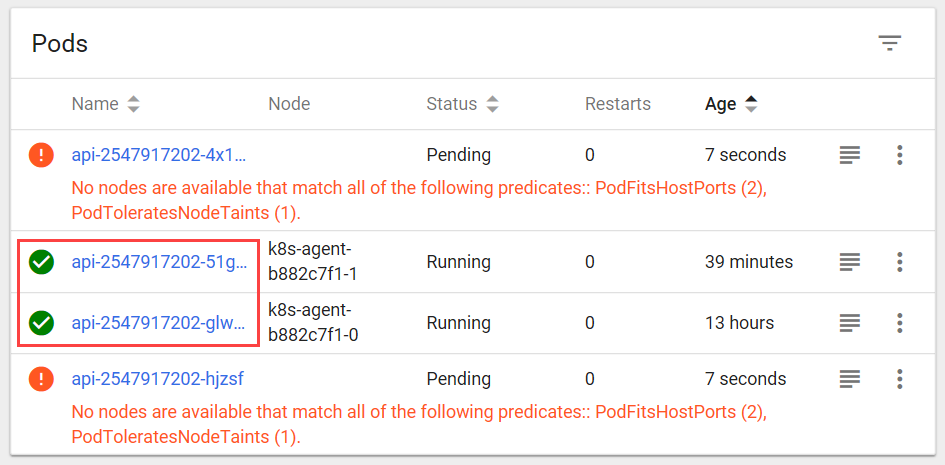
1. From the API deployment view, select **Scale** and from the dialog presented, enter 4 for the desired number of pods. Select **OK**.
2. From the navigation menu, select Workloads>Replica Sets. Select the api replica set and, from the Replica Set view, you will see that two pods cannot deploy.



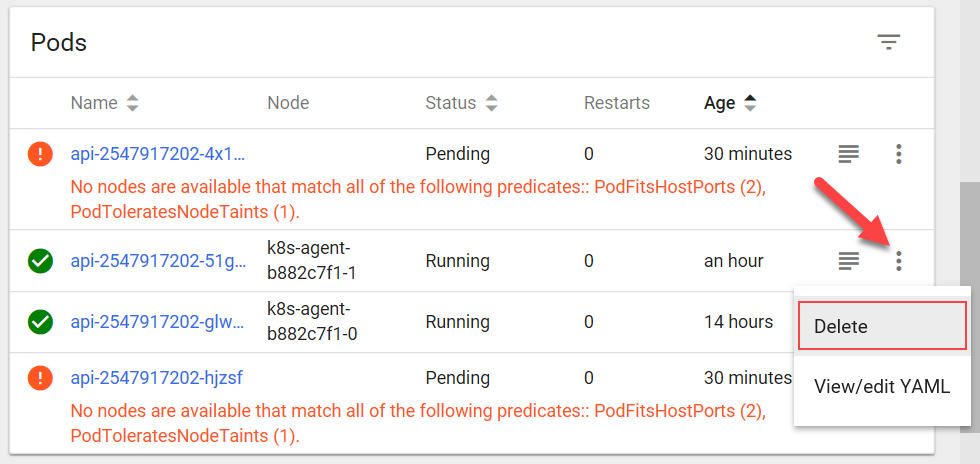
1. Return to the browser tab with the web application stats page loaded. Refresh the page over and over. You will not see any errors, but you will see the api host name change between the two api pod instances periodically. The task id and pid might also change between the two api pod instances.



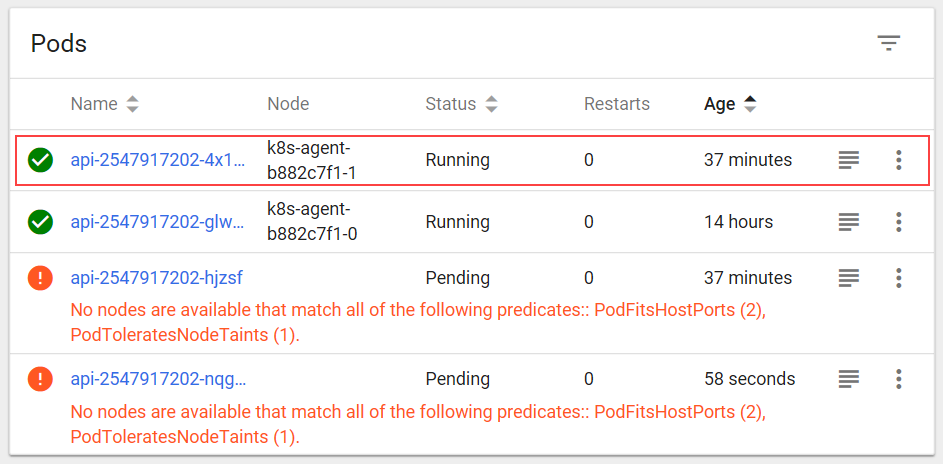
1. After refreshing enough times to see that the hostName value is changing, and the service remains healthy, return to the Replica Sets view for the API. From the navigation menu, select Replica Sets under Workloads and select the API replica set.
2. From this view, take note that the hostName value shown in the web application stats page matches the pod names for the pods that are running.



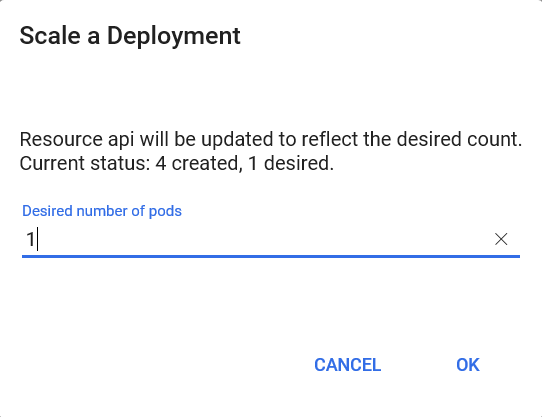
1. Note the remaining pods are still pending, since there are not enough port resources available to launch another instance. Make some room by deleting a running instance. Select the context menu and choose Delete for one of the healthy pods.



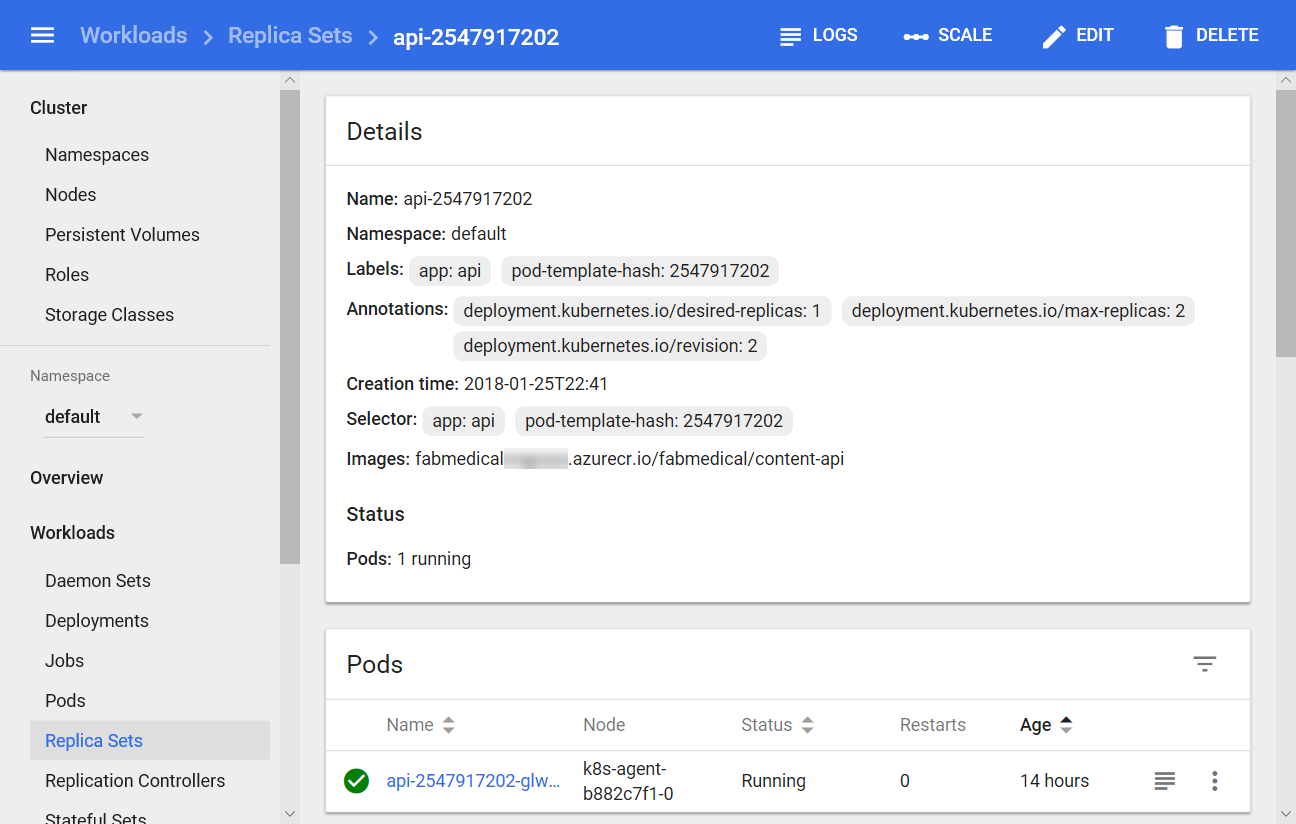
1. Once the running instance is gone, Kubernetes will be able to launch one of the pending instances. However, because you set the desired size of the deploy to 4, Kubernetes will add a new pending instance. Removing a running instance allowed a pending instance to start, but in the end, the number of pending and running instances is unchanged.



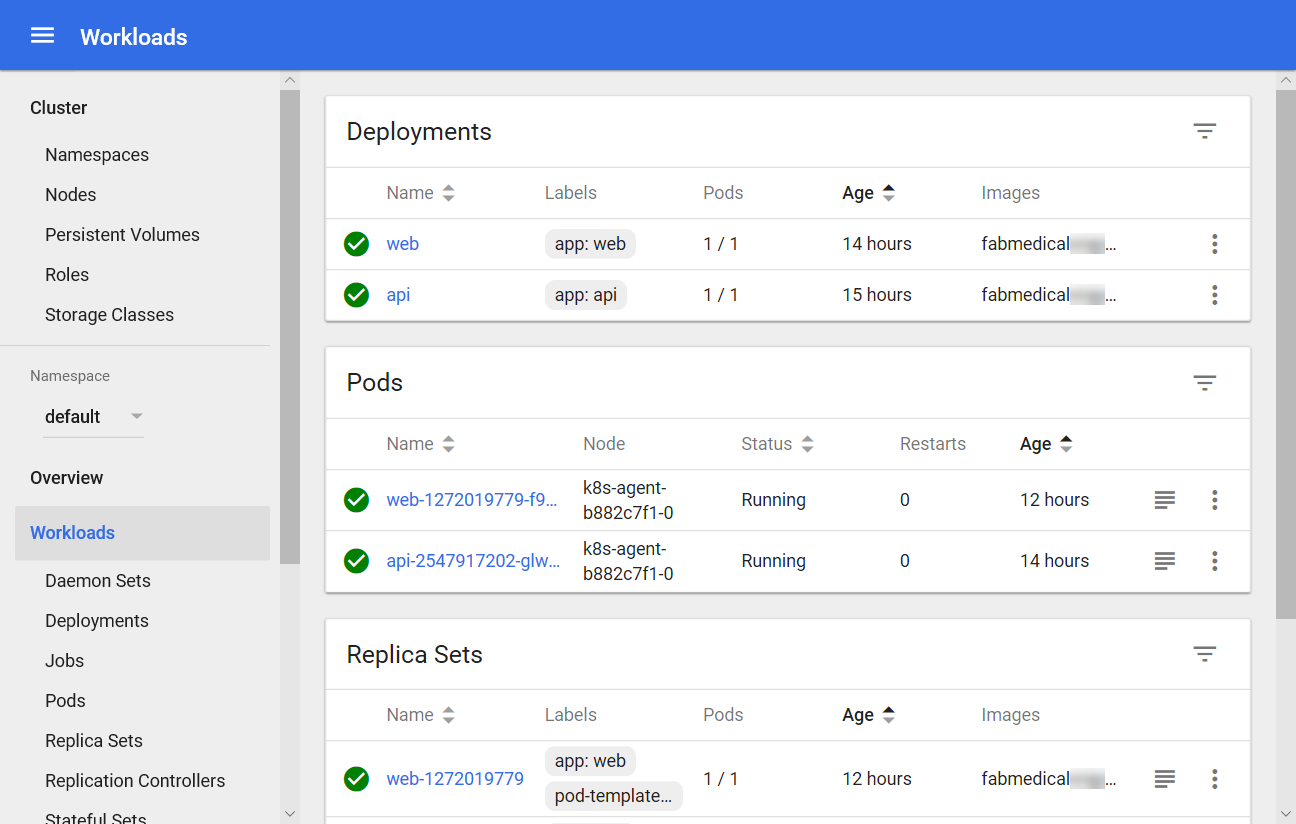
1. From the navigation menu, select Deployments under Workloads. From the view’s Deployments list select the API deployment.
2. From the API Deployment view, select Scale and enter 1 as the desired number of pods. Select OK.



1. Return to the web site’s stats.html page in the browser, and refresh while this is scaling down. You’ll notice that only one API host name shows up, even though you may still see several running pods in the API replica set view. Even though several pods are running, Kubernetes will no longer send traffic to the pods it has selected to scale down. In a few moments, only one pod will show in the API replica set view.



1. From the navigation menu, select Workloads. From this view, note that there is only one API pod now.



## Exercise 5: Setup load balancing and service discovery

**Duration**: 45 minutes

Until now there have been some restrictions to the scale properties of the service. In this exercise, you will configure a replica set to create pods that use dynamic port mappings to eliminate the port resource constraint during scale activities.

Kubernetes services can discover the ports assigned to each pod, allowing you to run multiple instances of the pod on the same agent node — something that is not possible when you configure a specific static port (such as 3001 for the API service).

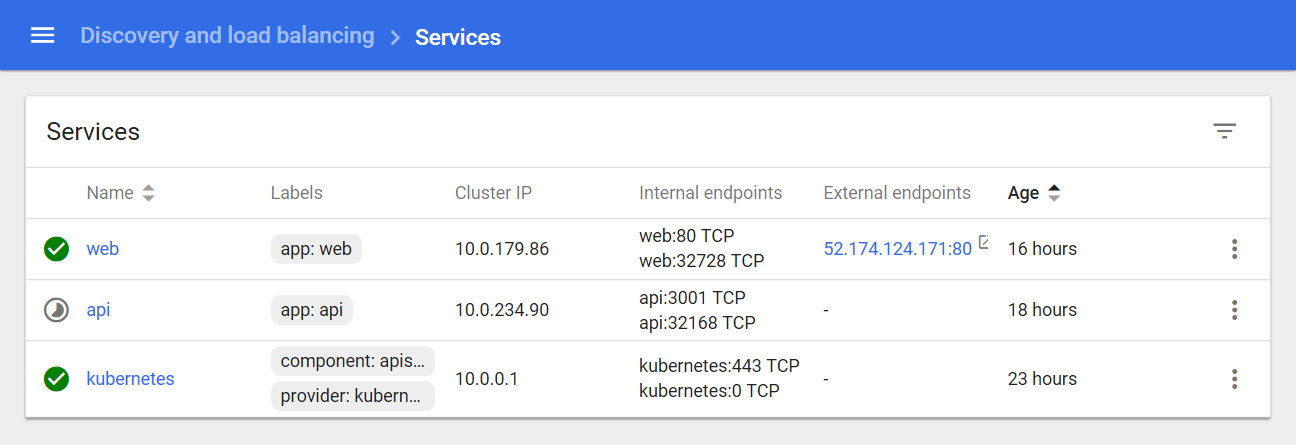
#### Task 1: Create a public load balancer for a service

Any external access to Kubernetes pods requires a load balancer. Because of this, you defined the web service with the type LoadBalancer in the YAML description, and you can access the web application using the public IP. In this task, you will change the API service type to LoadBalancer. When you change the type of the service to LoadBalancer, the ACS will create a public-facing load balancer with a public IP address.

1. From the navigation menu, select Services under Discovery and Load Balancing. From the view’s Services list, select the API service.
2. Select Edit.
3. From the Edit a Service dialog, scroll down to the type setting and change it from ClusterIP to LoadBalancer. Select Update to save the changes.

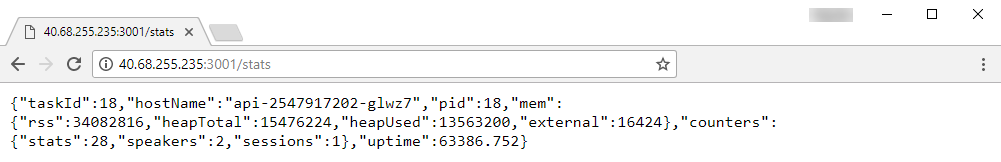


1. The API service will enter a pending state.



1. When the API service is ready again, it will have an external endpoint. You can select the hyperlink and navigate to the http://<external IP>:3001/stats path to see a JSON response. For example, considering the screenshot, the following URL would be used:
   * + 1. http://40.68.255.235:3001/stats

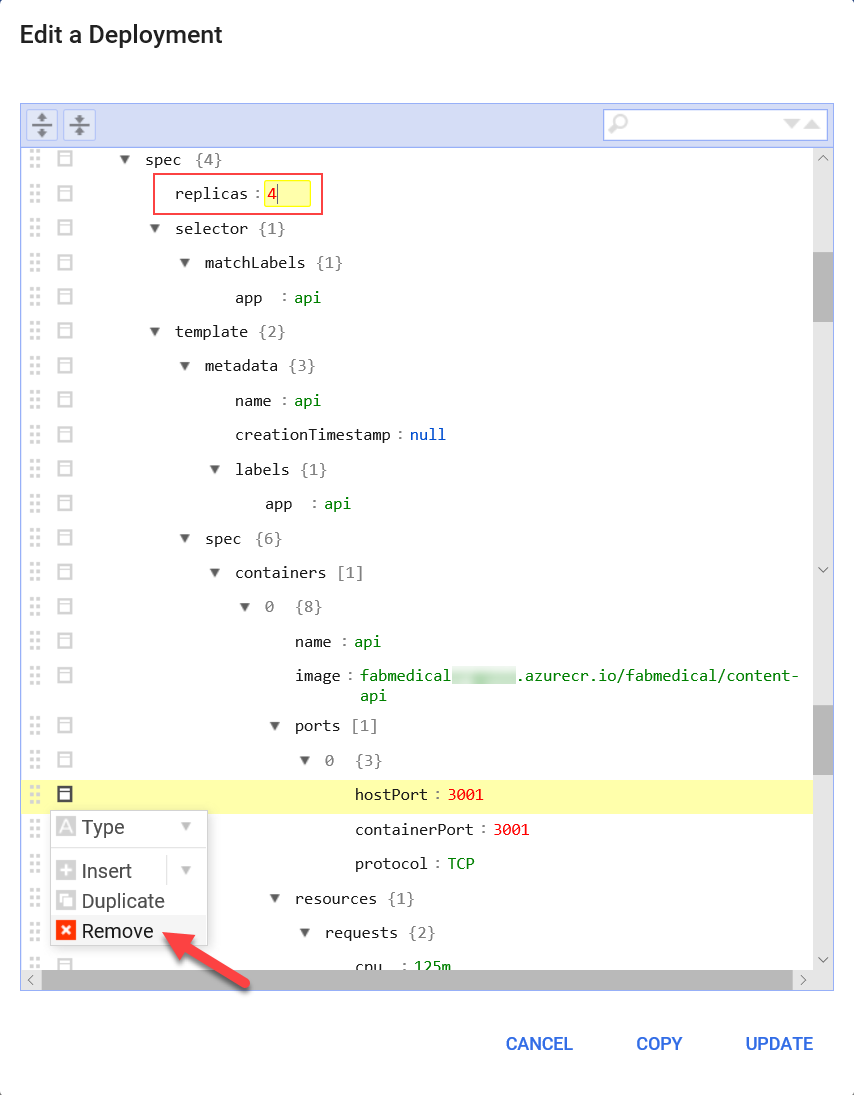


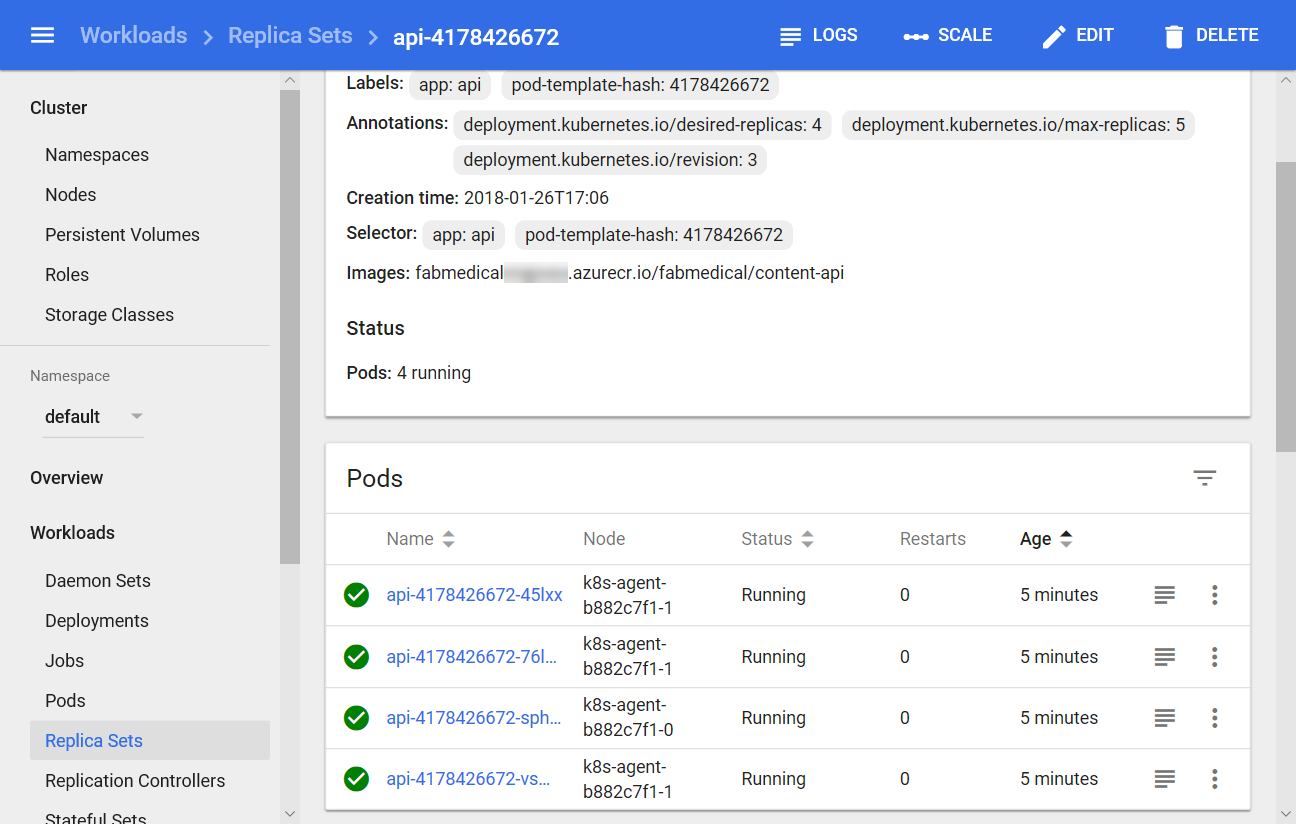


#### Task 2: Scale a service without port constraints

In this task, we will reconfigure the API deployment so that it will produce pods that choose a dynamic hostPort for routing from the Azure load balancer.

1. From the navigation menu select Deployments under Workloads. From the view’s Deployments list select the API deployment.
2. Select Edit.
3. From the Edit a Deployment dialog, do the following:
   1. Scroll to the first spec node that describes replicas as shown in the screenshot. Set the value for replicas to 4.
   2. Within the replicas spec, beneath the template node, find the API containers spec as shown in the screenshot. Remove the hostPort entry for the API container’s ports array by selecting the menu for the hostPort node and selecting Remove.

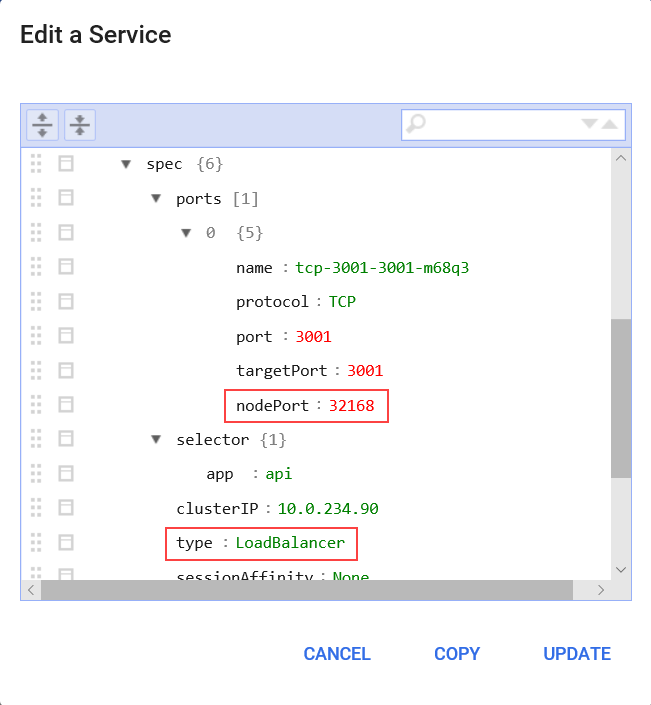


1. Select **Update**. New pods will now choose a dynamic port.
2. The API service can now scale to 4 pods since it is no longer constrained to an instance per node – a previous limitation while using port 3001. 
3. Return to the browser and refresh the API stats endpoint previously loaded. Now open a second browser and visit the API stats endpoint, in the second browser you should see a different hostName in the JSON produced by the stats endpoint.

#### Task 3: Update a service to support dynamic service discovery without a load balancer

The API service does not need a public endpoint to take advantage of the dynamic port discovery. Earlier in this lab you added a public load balancer with an external endpoint to directly observe dynamic discovery to learn about constraints. Now you will restore the API service to its original configuration in preparation for the following tasks that show off how seamlessly Kubernetes supports scaling services for internal endpoints as well.

1. From the navigation menu, select Services under Discovery and Load Balancing. From the view’s Services list select the API service.
2. Select **Edit**.
3. From the Edit a Service dialog, do the following:
   1. Scroll down to the *type* setting and change it from LoadBalancer to ClusterIP.
   2. Remove the *nodePort* setting.
   3. Select **Update**.



When the deployment update is complete, and the API service is ready, you will no longer be able to access /stats from the public endpoint. In the next task, you will update the web service to support dynamic discovery, but you will still be able to access the web application path at /stats.html as before and observe hostName and taskId changes to reflect the four API pods being called as you refresh.

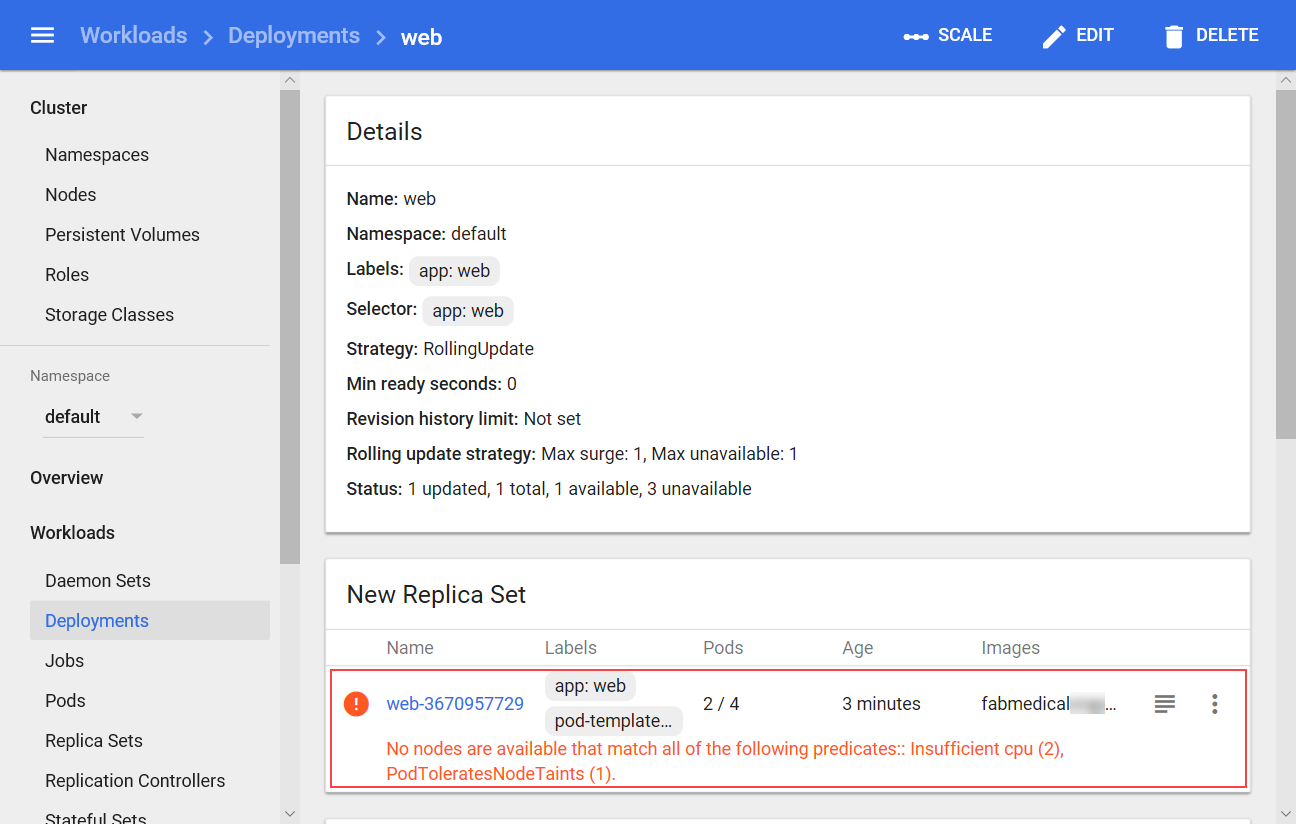
#### Task 4: Update an external service to support dynamic discovery with a load balancer

In this task, you will update the web service so that it supports dynamic discovery through the Azure load balancer.

1. From the navigation menu, select Deployments under Workloads. From the view’s Deployments list select the web deployment.
2. Select **Edit**.
3. From the Edit a Deployment dialog, scroll to the web containers spec as shown in the screenshot. Remove the hostPort entry for the web container’s ports array by selecting the menu for the hostPort node and selecting Remove.



1. Select **Update**.
2. From the web Deployments view, select **Scale**. From the dialog presented enter 4 as the desired number of pods and select **OK**.
3. Check the status of the scale out by refreshing the web deployments view. From the navigation menu, select Deployments from under Workloads. Select the web deployment and from this view you should see an error like that shown in the following screenshot.



Like the API deployment, the web deployment used a fixed *hostPort*, and your ability to scale was limited by the number of available agent nodes. However, after resolving this issue for the web service by removing the *hostPort* setting, the web deployment is still unable to scale past two pods due to CPU constraints. The deployment is requesting more CPU than the web application needs, so you will fix this constraint in the next task.

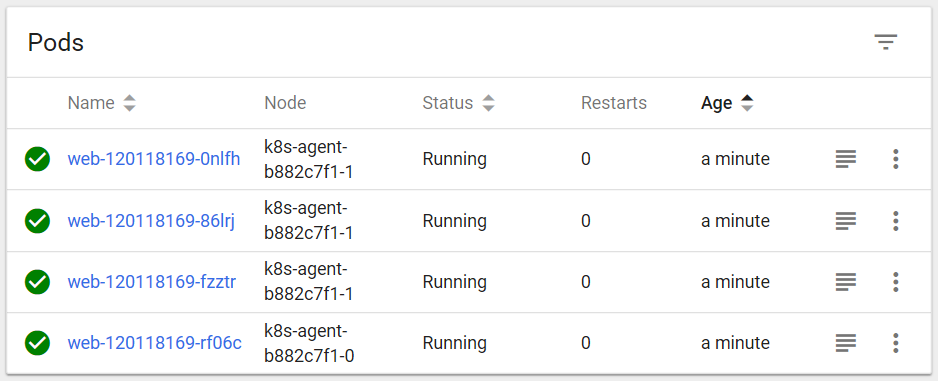
#### Task 5: Adjust CPU constraints to improve scale

In this task, you will modify the CPU requirements for the web service so that it can scale out to more instances.

1. From the navigation menu, select Deployments under Workloads. From the view’s Deployments list select the web deployment.
2. Select **Edit**.
3. From the Edit a Deployment dialog, find the *cpu* resource requirements for the web container. Change this value to “125m”.



1. Select **Update** to save the changes and update the deployment.
2. From the navigation menu, select Replica Sets under Workloads. From the view’s Replica Sets list select the web replica set.
3. When the deployment update completes, four web pods should be shown in running state.



1. Return to the browser tab with the web application loaded and refresh the stats page at /stats.html to watch the display update to reflect the different api pods by observing the host name refresh.

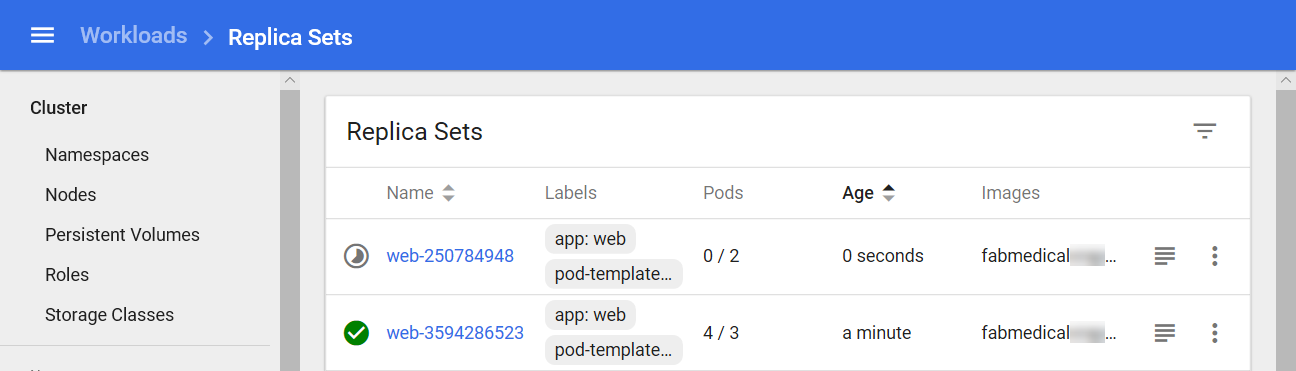
#### Task 6: Perform a rolling update

In this task, you will edit the web application source code and update the Docker image used by the deployment. Then you will perform a rolling update to demonstrate how to deploy a code change in a production scenario.

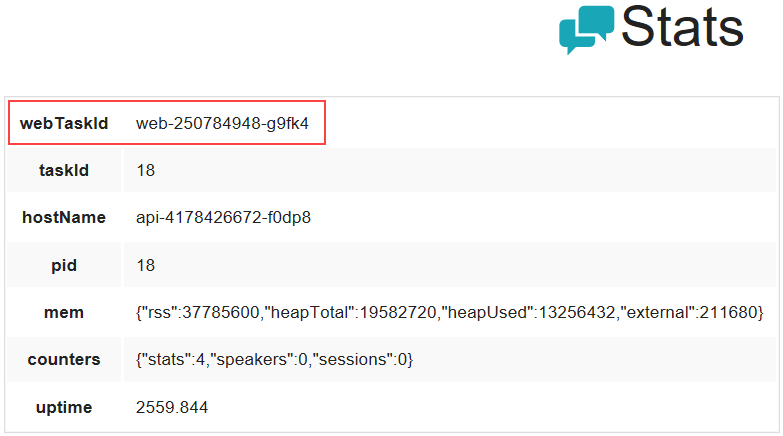
1. Connect to your build agent VM using ssh as you did earlier
2. From the command line, navigate to the FabMedical/content-web directory.
3. Open the server.js file using VI:
   * + 1. vi server.js
4. Enter insert mode by pressing <i>.
5. Scroll to the stats endpoint definition:
   * + 1. app.get(‘/api/stats’, function (req, res) {
       2. dataAccess.stats(function (error, data) {
       3. if (error) {
       4. return res.status(500).send(error);
       5. }
       6. else {
       7. data.webTaskId = process.pid;
       8. return res.status(200).send(data);
       9. }
       10. });
       11. });
6. Change the line that sets data.webTaskId to the following:
   * + 1. data.webTaskId = process.env.HOSTNAME;
7. Press the Escape key and type “:wq” and then press the Enter key to save and close the file.
   * + 1. <Esc>
       2. :wq
       3. <Enter>
8. Your current docker image for the web application should be “v1”. Verify this by running the command to view your docker images and look for two content-web images with “latest” and “v1” tags.

docker images

1. Build a new version of the docker image and then tag it with “v2”:
   * + 1. docker build -t [LOGINSERVER]/fabmedical/content-web .
       2. docker tag [LOGINSERVER]/fabmedical/content-web:latest [LOGINSERVER]/fabmedical/content-web:v2
2. Push these two new tagged images:
   * + 1. docker push [LOGINSERVER]/fabmedical/content-web:latest
       2. docker push [LOGINSERVER]/fabmedical/content-web:v2
3. Request a rolling update using this kubectl command:
   * + 1. kubectl set image deployment/web web=[LOGINSERVER]/fabmedical/content-web:v2
4. While this update runs, return the Kubernetes management dashboard in the browser.
5. From the navigation menu select Replica Sets under Workloads. From this view you will see a new replica set for web which may still be in the process of deploying (as shown below) or already fully deployed.



1. While the deployment is in progress, you can navigate to the web application and visit the stats page at /stats.html. Refresh the page as the rolling update executes. Once the update completes, you will see the webTaskId begins to update correctly to show the hostName of the web pod, as shown here.



1. Open a second browser to view the web application stats page and note that the webTaskId is different for each browser as requests are routed among the available web pods. Notice also that the hostName of the API pod changes from time to time as well when the browser is refreshed.