It's easy to create a function in R, but what if you want to call that function from a different application, with the scale to support a large number of simultaneous requests? This article shows how you can deploy an R fitted model as a [Plumber web service](https://www.rplumber.io/) in Kubernetes, using [Azure Container Registry](https://docs.microsoft.com/azure/container-registry/?WT.mc_id=Revolutions-blog-davidsmi) (ACR) and [Azure Kubernetes Service](https://docs.microsoft.com/en-us/azure/aks/?WT.mc_id=Revolutions-blog-davidsmi) (AKS). We use the  AzureContainers package to create the necessary resources and deploy the service.

**Fit the model**

We’ll fit a simple model for illustrative purposes, using the [Boston housing dataset](https://www.rdocumentation.org/packages/MASS/versions/7.3-51.1/topics/Boston?tap_a=5644-dce66f&tap_s=10907-287229) (which ships with R in the MASS package). To make the deployment process more interesting, the model we fit will be a random forest, using the randomForest package. This is *not* part of R, so we’ll have to install it from CRAN.

data(Boston, package="MASS")

install.packages("randomForest")

library(randomForest)

# train a model for median house price

bos\_rf <- randomForest(medv ~ ., data=Boston, ntree=100)

# save the model

saveRDS(bos.rf, "bos\_rf.rds")

**Scoring script for plumber**

Now that we have the model, we also need a script to obtain predicted values from it given a set of inputs:

# save as bos\_rf\_score.R

bos\_rf <- readRDS("bos\_rf.rds")

library(randomForest)

#\* @param df data frame of variables

#\* @post /score

function(req, df)

{

df <- as.data.frame(df)

predict(bos\_rf, df)

}

This is fairly straightforward, but the comments may require some explanation. They are plumber annotations that tell it to call the function if the server receives a HTTP POST request with the path /score, and query parameter df. The value of the df parameter is then converted to a data frame, and passed to the randomForest predict method. For a fuller description of how Plumber works.

**Create a Dockerfile**

Let’s package up the model and the scoring script into a Docker image. A Dockerfile to do this is shown below. This uses the base image supplied by Plumber (trestletech/plumber), installs randomForest, and then adds the model and the above scoring script. Finally, it runs the code that will start the server and listen on port 8000.

# example Dockerfile to expose a plumber service

FROM trestletech/plumber

# install the randomForest package

RUN R -e 'install.packages(c("randomForest"))'

# copy model and scoring script

RUN mkdir /data

COPY bos\_rf.rds /data

COPY bos\_rf\_score.R /data

WORKDIR /data

# plumb and run server

EXPOSE 8000

ENTRYPOINT ["R", "-e", \

"pr <- plumber::plumb('/data/bos\_rf\_score.R'); \

pr$run(host='0.0.0.0', port=8000)"]

**Build and upload the image**

The code to store our image on Azure Container Registry is as follows. This calls [AzureRMR](https://cran.r-project.org/package=AzureRMR) to login to Resource Manager, creates an Azure Container Registry resource (a Docker registry hosted in Azure), and then pushes the image to the registry.

If this is the first time you are using AzureRMR, you’ll have to create a service principal first. For more information on how to do this.

library(AzureContainers)

az <- AzureRMR::az\_rm$new(

tenant="[myaadtenant.onmicrosoft.com](http://myaadtenant.onmicrosoft.com)",

app="app\_id",

password="password")

# create a resource group for our deployments

deployresgrp <- az$

get\_subscription("subscription\_id")$

create\_resource\_group("deployresgrp", location="australiaeast")

# create container registry

deployreg\_svc <- deployresgrp$create\_acr("deployreg")

# build image 'bos\_rf'

call\_docker("build -t bos\_rf .")

# upload the image to Azure

deployreg <- deployreg\_svc$get\_docker\_registry()

deployreg$push("bos\_rf")

If you run this code, you should see a lot of output indicating that R is downloading, compiling and installing randomForest, and finally that the image is being pushed to Azure. (You will see this output even if your machine already has the randomForest package installed. This is because the package is being installed to the R session *inside the container*, which is distinct from the one running the code shown here.)

All docker calls in AzureContainers, like the one to build the image, return the actual docker commandline as the cmdline attribute of the (invisible) returned value. In this case, the commandline is docker build -t bos\_rf . Similarly, the push() method actually involves two Docker calls, one to retag the image, and the second to do the actual pushing; the returned value in this case will be a 2-component list with the command lines being docker tag bos\_rf [deployreg.azurecr.io/bos\_rf](http://deployreg.azurecr.io/bos_rf) and docker push [deployreg.azurecr.io/bos\_rf](http://deployreg.azurecr.io/bos_rf).

**Deploy to a Kubernetes cluster**

The code to create an AKS resource (a managed Kubernetes cluster in Azure) is quite simple:

# create a Kubernetes cluster with 2 nodes, running Linux

deployclus\_svc <- deployresgrp$create\_aks("deployclus",

agent\_pools=aks\_pools("pool1", 2))

Creating a Kubernetes cluster can take several minutes. By default, the create\_aks() method will wait until the cluster provisioning is complete before it returns.

Having created the cluster, we can deploy our model and create a service. We’ll use a YAML configuration file to specify the details for the deployment and service API.

apiVersion: extensions/v1beta1

kind: Deployment

metadata:

name: bos-rf

spec:

replicas: 1

template:

metadata:

labels:

app: bos-rf

spec:

containers:

- name: bos-rf

image: [deployreg.azurecr.io/bos\_rf](http://deployreg.azurecr.io/bos_rf)

ports:

- containerPort: 8000

resources:

requests:

cpu: 250m

limits:

cpu: 500m

imagePullSecrets:

- name: [deployreg.azurecr.io](http://deployreg.azurecr.io)

---

apiVersion: v1

kind: Service

metadata:

name: bos-rf-svc

spec:

selector:

app: bos-rf

type: LoadBalancer

ports:

- protocol: TCP

port: 8000

The following code will obtain the cluster endpoint from the AKS resource and then deploy the image and service to the cluster. The configuration details for the deployclus cluster are stored in a file located in the R temporary directory; all of the cluster’s methods will use this file. Unless told otherwise, AzureContainers does not touch your default Kubernetes configuration (~/kube/config).

# get the cluster endpoint

deployclus <- deployclus\_svc$get\_cluster()

# pass registry authentication details to the cluster

deployclus$create\_registry\_secret(deployreg,

email="[me@example.com](mailto:me@example.com)")

# create and start the service

deployclus$create("bos\_rf.yaml")

To check on the progress of the deployment, run the get() methods specifying the type and name of the resource to get information on. As with Docker, these correspond to calls to the kubectl commandline tool, and again, the actual commandline is stored as the cmdline attribute of the returned value.

deployclus$get("deployment bos-rf")

#> Kubernetes operation: get deployment bos-rf --kubeconfig=".../kubeconfigxxxx"

#> NAME DESIRED CURRENT UP-TO-DATE AVAILABLE AGE

#> bos-rf 1 1 1 1 5m

deployclus$get("service bos-rf-svc")

#> Kubernetes operation: get service bos-rf-svc --kubeconfig=".../kubeconfigxxxx"

#> NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

#> bos-rf-svc LoadBalancer 10.0.8.189 52.187.249.58 8000:32276/TCP 5m

Once the service is up and running, as indicated by the presence of an external IP in the service details, let’s test it with a HTTP request. The response should look like this.

response <- httr::POST("<http://52.187.249.58:8000/score>",

body=list(df=MASS::Boston[1:10,]), encode="json")

httr::content(response, simplifyVector=TRUE)

#> [1] 25.9269 22.0636 34.1876 33.7737 34.8081 27.6394 21.8007 22.3577 16.7812 18.9785

Finally, once we are done, we can tear down the service and deployment:

deployclus$delete("service", "bos-rf-svc")

deployclus$delete("deployment", "bos-rf")

And if required, we can also delete all the resources created here, by simply deleting the resource group (AzureContainers will prompt you for confirmation):

deployresgrp$delete()