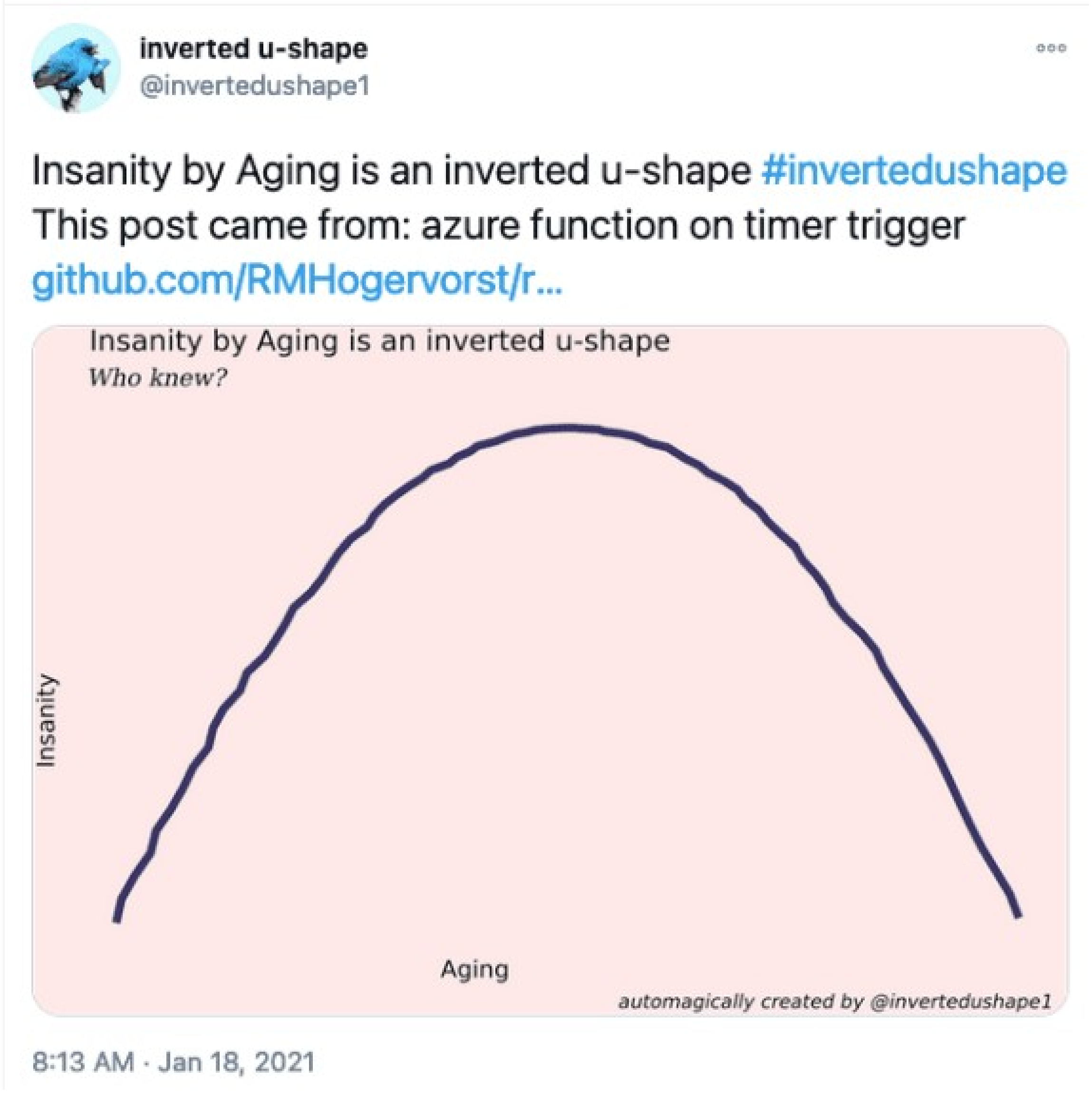
What the app does

The R-script, in this post: generates random data, selects random phrases, creates an image and publishes that picture to twitter under the name @invertedushape1.


inverted u-shape
@invertedushape1

Insanity by Aging is an inverted u-shape #invertedushape
This post came from: azure function on timer trigger
github.com/RMHogervorst/r...

Insanity by Aging is an inverted u-shape
Who knew?
Insanity
Aging
automagically created by @invertedushape1

8:13 AM · Jan 18, 2021

I chose this tweet-example because it is slightly silly, and this script:

- has visible output
- makes use of {renv} for consistent package-versions
- talks to an outside API on the internet
- deals with credentials that need to be kept safe
- logs what it does
- is entirely contained in a docker container

In my opinion that covers quite some relevant parts of modern cloud applications!

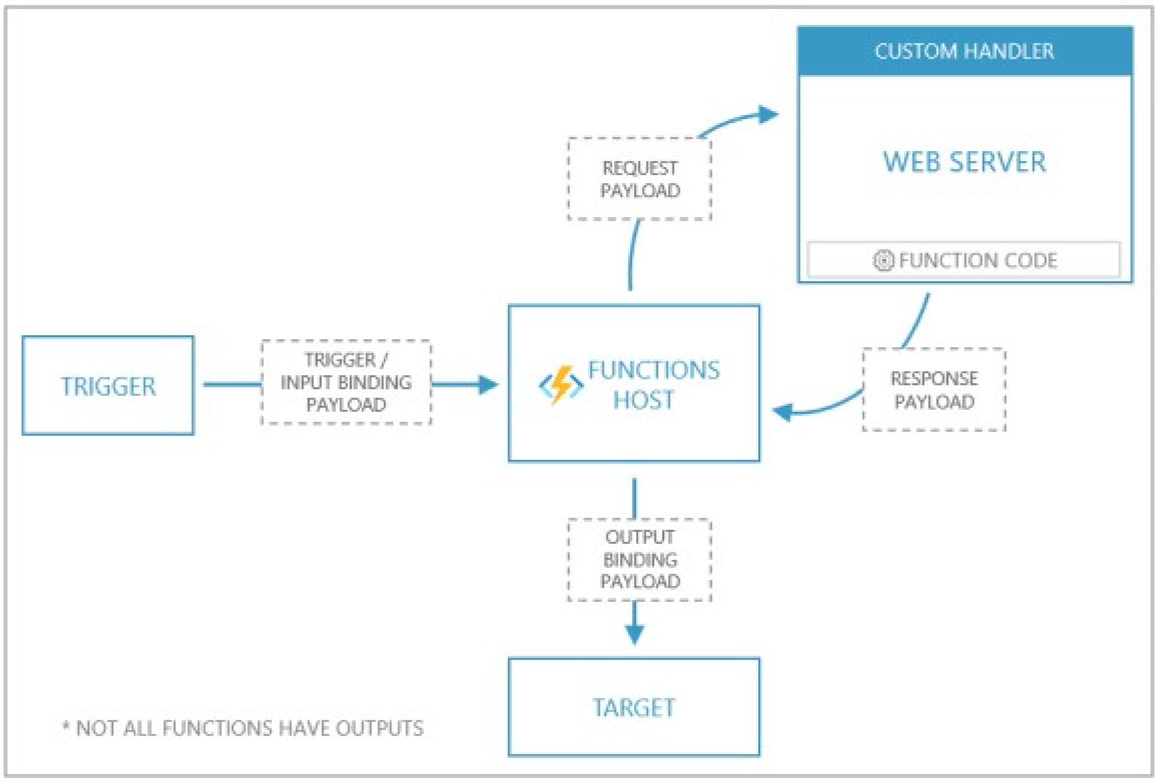
When to use serverless

The idea of Serverless or ‘Function as a Service’ is that you only write the code that you want to execute and you only pay for execution of the function. You don’t have to maintain servers or other infrastructure. Ideally you write the function, dump it into the serverless infrastructure of your choice: AWS has ‘lambda’, Azure has ‘Azure Functions’ and Google has ‘GCP Cloud Functions’. So there ARE servers and underlying infrastructure, but YOU don’t have to maintain the infrastructure nor think about infrastructure.

triggers

Not only in Azure, but everywhere, your custom code runs when it is triggered. The cloud service sends the trigger, and including some data about the trigger, to your function and your function does something with that information.

Azure gives the following image as an overview:



What is a bit misleading in this picture is that both the functions host and custom handler are implemented inside the same docker container. We create one container, based on the Microsoft docker-image, with a web server (in R) that responds to requests.

The easiest and most common trigger is a webrequest. For instance, a new user account is created, or a prediction is requested. But there are many more triggers for instance: database record update, new item in a cue, incoming webhook, new item in blob storage or new message in a kafka topic. In my case I’m using the most dumb trigger imaginable: a time trigger. My function should run, once a day. This is also known as a CRON job (CRON is the time-based scheduler in unix and so it is very widely known, and still used everywhere). *In a way I’m abusing the system because I just want a short script to run at set times, and this is a cheap way to do it. It is the ultimate pay-only-for-what-you-use offering in the cloud.*

Machinelearning in serverless?

Training a machine learning model is absolutely **not** a good task for serverless. You are severely limited in CPU, memory and time. So you could train in a larger machine, save the model and serve predictions in a serverless way, in fact David Smith shows exactly that example in his blogpost.

Serverless R for Azure

To make a serverless application for R in Azure you need:

- a script that does your action
- resources in azure: storage, function itself, and a resourcegroup
- a docker container based on the Microsoft-supplied function-app docker image
- to respond to triggers (more about that later)
- to test if everything works locally

Overview of the process

Again, if you want to create something like what I create in this post, I really advise you to read and follow the tutorial by David Smith first and modify afterwards with some help from this post.

Azure functions does not support R out of the box (but has an open system where you can run docker containers and so it supports all languages of the world). Most importantly, you need to create something that responds to API requests. David and I use {plumber}, but in theory you could build the same thing with any other framework (See below in references).

For this example I have 2 scripts:

- a script that defines the plumber server and responds to requests
- a script that defines the routes and executes something I actually care about

steps to take

- setup
- Create some json files for configuration and Set the schedule
- test the function locally
- create a dockerfile
- test the function in the docker container locally
- push everything into Azure
- Dealing with secrets
- auto deployment

Setup

- Make sure you actually have an Azure account
- Make sure you have installed Azure functions core tools AND azure CLI AND docker instructions here

Like David I'm going to use dockerhub, so make sure you have an account (dockerID) or use a different registry where you can push to.

- log in azure tools `az login` (this opens a new browser window)
- If you enabled MFA in your azure account (Good job!) you also need to do an extra step once but you get suggestions in your command line.
- make sure docker is running
- log in to docker hub `docker login` (or your other container registry of choice)

Create configuration files

Now, in the terminal (commandline), I'm going to the folder where I have my R project.

- Create a local custom functions project: `func init --worker-runtime custom`

```
    --docker
```

This creates a .gitignore file, local.settings.json, .vscode/extensions.json. dockerfile. and docker ignore. Super useful, but not complete. **If you use environmental variables** in this project to maintain your secrets (good job!) you probably saved them in an .Renviron file in this project. You don't want that file send to your public repository nor included in a public dockerfile. So let's add some lines to the files.

- In the .gitignore file, and .dockerignore file add:

```
## R files to ignore
.Renviron
.Rproj.user
.Rhistory
.RData
.Ruserdata
```

- Create the function configuration with a template `func new --name --template "timer trigger"` (using the timer trigger template, *if you need something else, look it up through the links in the reference section at the bottom of this page*)

This creates a folder with the name you just gave and inside that folder you find a function.json that defines the 'bindings' in this case a timer trigger with a schedule. The schedule is defined like in CRON, for instance: 'once a day at 2300 hours' ("0 23 * * *")

- Find the name of your Rscript that creates the server (that responds to requests) or make a script and give it a name, I'm calling mine handler.R.

- I'm going to modify the host.json to call my Rscript when it is triggered.

Specifically this part in the json:

```
"customHandler": {
"description": {
"defaultExecutablePath": "Rscript",
"workingDirectory": "",
"arguments": [
"handler.R"
]
}
}
```

I'm telling it to call the executable Rscript, with the argument `handler.R` So it is running the command `Rscript handler.R` .

Your folder could then look something like mine:

```
├── .Renviron
├── .Rhistory
├── .Rprofile
├── .dockerignore
├── .gitignore
├── CRONtrigger
│   └── function.json
├── Dockerfile
├── README.md
├── handle-this.R
├── handler.R
├── host.json
├── local.settings.json
├── plumber_run_script.Rproj
├── renv
│   ├── activate.R
│   ├── library
│   ├── settings.dcf
│   └── staging
├── renv.lock
```

- .Renviron (for secrets)
- .Rprofile (for activating renv)
- .dockerignore and .gitignore to ignore secrets
- I'm calling my function 'CRONtrigger' and so there is a folder named CRONtrigger with a 'function.json' inside it
- I have 2 R scripts, handler.R which creates the server and 'handle-this.R' that is called by the server and does all the work
- there is also a host.json that contains settings, like logging levels. and what executible to call to run the custom-handler: Rscript in my case.
- the folder renv contains the library and is ignored by the docker files
- the renv.lock file contains the curent exact versions of packages I'm using. It gets updated everytime you call `renv::snapshot()`

Test the function locally

Make sure all required packages are installed in your local library. I'm using {renv} for all my projects. Check if everything works by running the script with source("handle-this.R") or outside Rstudio with the terminal: `Rscript handle-this.R` . Now I can test the functionality of the serverless framework with `func start`.

create a dockerfile

Now the script works I continue with `renv::snapshot()` to make sure the packages are up to date in the 'renv.lock' file. I cannot use a rocker container because we need a special azure-functions container that knows how to talk to the azure functions framework.

The Dockerfile that David created is really nice, but I modified it a bit to deal with renv. Crucially (and something I forgot the first time) the system only works if all the important files are placed in '/home/site/wwwroot' in the container.

- I installed some system requirements I knew I needed from previous docker experiments with this app.
- I create a renv directory and install renv (as seen on the renv website)
- I copy the 'renv.lock' file to the container
- I install all the required packages `renv::restore()`
- I copy all the setting jsons
- I copy the Rscript

Example Dockerfile at github or below:

```
FROM mcr.microsoft.com/azure-functions/dotnet:3.0
ENV AzureWebJobsScriptRoot=/home/site/wwwroot \
 AzureFunctionsJobHost__Logging__Console__IsEnabled=true

RUN apt update && \
 apt install -y r-base
RUN apt-get install -y --no-install-recommends \
 libcurl4-openssl-dev \
 libssl-dev libxt6 && \
 mkdir -p ~/.local/share/renv && \
 R -e "install.packages('renv', repos='http://cran.rstudio.com/')"
### splitted the copying into parts so the rebuiliding times are
quicker
COPY renv.lock /home/site/wwwroot/renv.lock
WORKDIR /home/site/wwwroot
RUN R -e "renv::restore(prompt=FALSE)"
COPY TimerRtrigger/ /home/site/wwwroot/TimerRtrigger
COPY host.json /home/site/wwwroot/host.json
COPY run_job.R /home/site/wwwroot/run_job.R
```

When you're done with developing it can be beneficial to combine multiple statements in the Dockerfile because this can reduce the size of the images.

Test the function in the docker container locally

- Build the container `docker build --tag /:v1.0.0 .`
- Test the container, supplying the .Renviron secrets `docker run --env-file .Renviron -p 8080:80 -it /:v1.0.0`

to see what the parts of the command mean, unhide this paragraph This command need some explanation. Docker run /:v1.0.0 runs the container, if you need to open some port on the container use `ip:hostPort:containerPort` For interactive processes (we want to see what is happening in this container now), you must use `-i` `-t` together in order to allocate a tty for the container process. `-i -t` is often written `-it`. Because my secrets live in a .Renviron file I can pass that entire file to the docker container to supply to the process. for more info see [the docker documentation](https://docs.docker.com/engine/reference/run/).

Because I'm using the timer trigger in Azure I need to tell the function where to find storage. you can supply the actual credentials of azure or use docker-compose to spin up your container and the azure storage simulator (azurite). In my previous post I explain how.

push everything to Azure

I'm going to push the docker image to docker hub and tell Azure functions where to find the image.

- push docker image to docker hub `docker push /:v1.0.0`
- find the name of a region that support azure functions premium plan linux `az account list-location -o table` (which is almost all of them)

I'm choosing one in my region, but it really doesn't matter that much for my particular usecase: I'm creating something in the runtime and pushing it to twitter. Twitter has worldwide coverage and so I think it doesn't really matter where my function lives. So maybe I should choose a very cheap location? (I did a quick exploration and it seems it costs the same everywhere €0.169 per million executions).

For any other usecase you want to be close to the place your pulling data from or pushing data to so that latency is minimal (if you do database inserts from a function that runs in Germany to a database that lives in Australia, there will be some delay, and thus higher cost).

- Define a unique name for resource group, also choose a name for your function and a storage account to host assets (I copied this from David's tutorial, he defined first the variables and later calls the variables).

Defining variables:

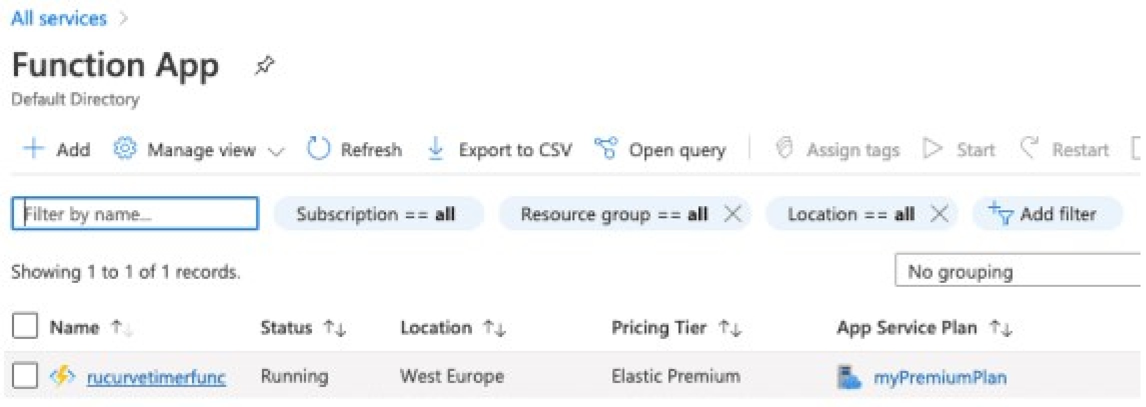
```
FR_LOC="westeurope"
FR_RG="R-u-curve-timer"
FR_FUNCTION="rucurvetimerfunc"
FR_STORAGE="rucurvetimerstrg"
```

Creating resources:

```
az group create --name $FR_RG --location $FR_LOC
az storage account create --name $FR_STORAGE --location $FR_LOC
--resource-group $FR_RG --sku Standard_LRS
az functionapp plan create --resource-group $FR_RG --name myPremiumPlan
--location $FR_LOC --number-of-workers 1 --sku EP1 --is-linux
az functionapp create --functions-version 2 --name $FR_FUNCTION
--storage-account $FR_STORAGE --resource-group $FR_RG --plan
myPremiumPlan --runtime custom --deployment-container-image-name
/:v1.0.0
storageConnectionString=$(az storage account show-connection-string
--resource-group $FR_RG --name $FR_STORAGE --query connectionString
--output tsv)
az functionapp config appsettings set --name $FR_FUNCTION --resource-
group $FR_RG --settings AzureWebJobsStorage=$storageConnectionString
```

These commands took some time for me to execute. Every command returns a json with status and information.

If everything worked out, in your azure account you see the function we just created over the commandline.

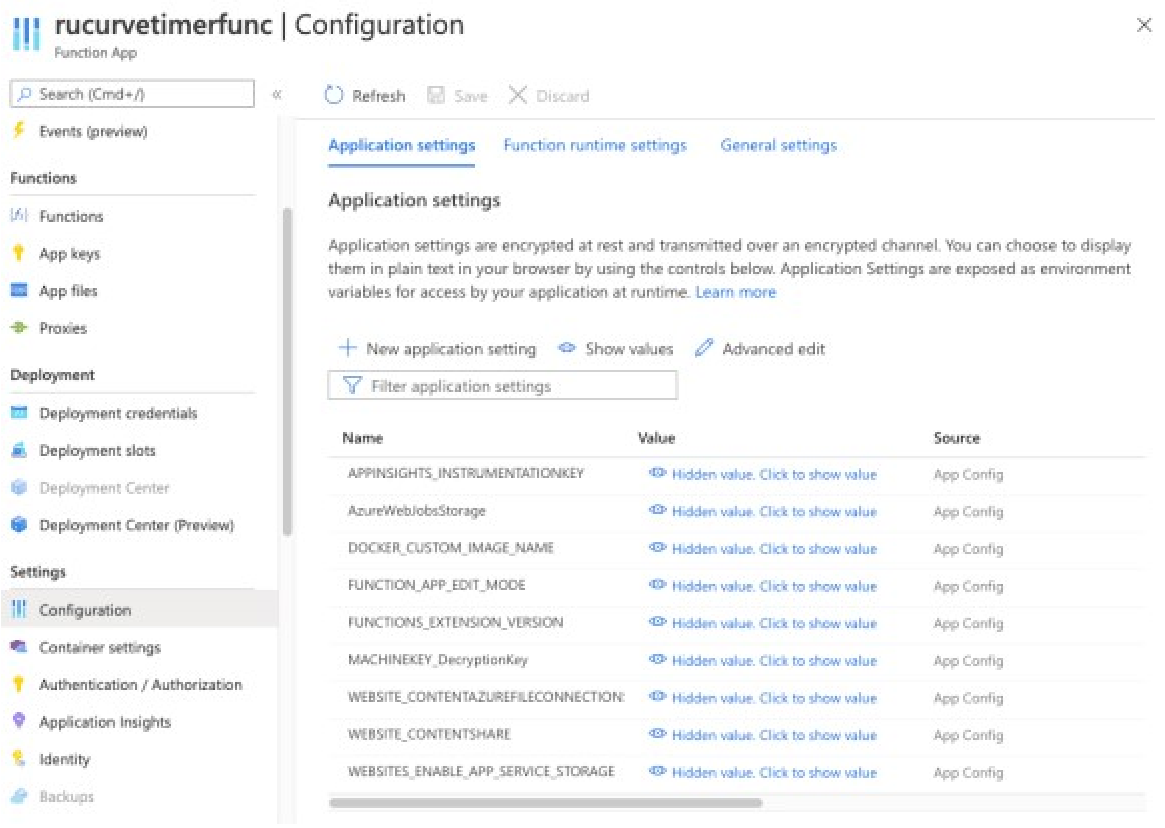


Showing 1 to 1 of 1 records.

Name	Status	Location	Pricing Tier	App Service Plan
rucurvetimerfunc	Running	West Europe	Elastic Premium	myPremiumPlan

Dealing with Secrets

My app needs to talk to Twitter, so it needs some secrets to authenticate against Twitter. However you should never(!) hardcode these secrets in the script: if you share the script, place it in a public repository or if the secrets are changed you need to modify the script. In my script I call `Sys.getenv("VARNAME")`, so that the script looks around and searches for an environmental variable (with the name VARNAME) that is set. Locally I have an .Renviron file that is read by R during startup and reads in the secrets as env variables. But you can also set these variables in other ways, for docker I supply them through the commandline interface and in Azure we set them in the 'settings' tab of the Function App. As you can see in the image below several variables are already set, and I added new ones for my twitter keys.



Application settings

Application settings are encrypted at rest and transmitted over an encrypted channel. You can choose to display them in plain text in your browser by using the controls below. Application Settings are exposed as environment variables for access by your application at runtime. Learn more

Name	Value	Source
APPINSIGHTS_INSTRUMENTATIONKEY	Hidden value. Click to show value	App Config
AzureWebJobsStorage	Hidden value. Click to show value	App Config
DOCKER_CUSTOM_IMAGE_NAME	Hidden value. Click to show value	App Config
FUNCTION_APP_EDIT_MODE	Hidden value. Click to show value	App Config
FUNCTIONS_EXTENSION_VERSION	Hidden value. Click to show value	App Config
MACHINEKEY_DecryptionKey	Hidden value. Click to show value	App Config
WEBSITE_CONTENTAZUREFILECONNECTION:	Hidden value. Click to show value	App Config
WEBSITE_CONTENTSHARE	Hidden value. Click to show value	App Config
WEBSITES_ENABLE_APP_SERVICE_STORAGE	Hidden value. Click to show value	App Config

Auto deployment

You can set up a hook in dockerhub that sends a signal to Azure to rebuild the function