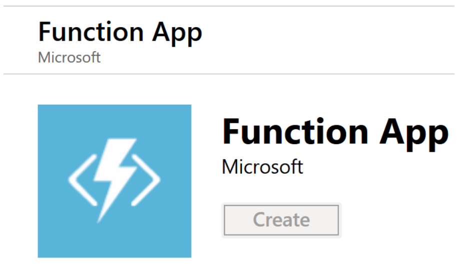
Azure Functions and Docker

Azure functions consist normally of 2 parts:

1. Azure Function App (container)
2. Function (code)

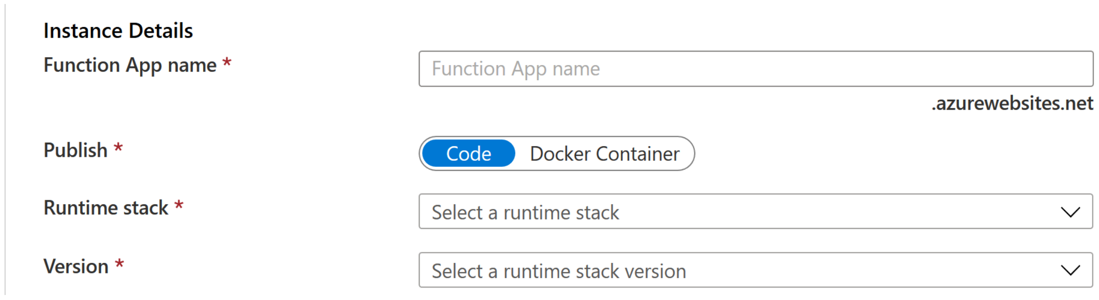


Until recently when creating a new function app in Azure, you need to choose which runtime you want to use:

- .Net core
- Node.js
- Python
- Java
- Powershell Core

And which version of that runtime.

But sometime ago there was a different option added: Docker container.



Instance Details

Function App name *	Function App name .azurewebsites.net
Publish *	Code / Docker Container
Runtime stack *	Select a runtime stack
Version *	Select a runtime stack version

If you choose Docker container you can not select the runtime any more.

And another difference is that can only select Linux as the operation system for your runtime stack. For regular functions the default is Windows.

And the third difference is that you can not choose consumption plan, only Premium and App service plan.

> **Note that you choose the runtime stack on which to deploy your functions, not the functions them self! In that respect it is different from for example the Fn Project which is used by Oracle for its cloud functions.**

But why / when would you choose to use Docker containers as runtime stack for Azure functions?

A reason could be that you have a dependency on a certain version of the language runtime stack or some specific configuration which is not available out of the box. Maybe you have the need for some external tools you want to have available for your function(s) and so not want to install them first (if that is possible) and every time you (re) create the function app.

Another advantage is, because the runtime is Docker, you now can run your functions anywhere you like as long as you have a Docker runtime available. So you could even run Azure Functions on premise.

Small workshop

Prepare

Before starting creating a new Azure function, you need to have locally installed:

- Visual Studio Code: https://code.visualstudio.com/
- Node.js v10 or v12 and NPM: https://nodejs.org/en/
- Docker: https://docs.docker.com/install/
- Azure functions runtime 3.x: https://docs.microsoft.com/en-us/azure/azure-functions/functions-run-local?tabs=windows%2Ccsharp%2Cbash
- Azure subscription and Azure CLI when deploying to Azure: https://docs.microsoft.com/en-us/cli/azure/install-azure-cli?view=azure-cli-latest
- If you want to save your Docker image, you need access to a Docker registry like Docker Hub or an Azure Container Registry

After installation of Visual Studio Code and Node.js open a terminal or command window and install Azure functions runtime v3 (when not yet done) and type:

```
npm install -g azure-functions-core-tools@3
```

run `func --version` to check the version.

Local functions project

Create a new folder for your Azure Functions project and navigate to it in a terminal or command window

And type:

```
func init --docker
```

Choose for worker runtime option 2: node and as language: javascript

```
$ func init --docker
Select a number for worker runtime:
1. dotnet
2. node
3. python
4. powershell
Choose option: 2
```

```
node
Select a number for language:
1. javascript
2. typescript
Choose option: 1
javascript
Writing package.json
Writing .gitignore
Writing host.json
Writing local.settings.json
Writing /home/henkjan/testfunc/.vscode/extensions.json
Writing Dockerfile
Writing .dockerignore
```

Now add a sample function by typing:

```
func new --name MySampleFunction --template "HTTP trigger"
```

If you want to see all the template options, type: `func new --name MySampleFunction`.

You can already test the sample function by typing:

```
func start
```

This will start the local Azure Functions runtime host.

Open a browser and navigate to: http://localhost:7071/api/MySampleFunction

You will see a message in your browser that you need to pass a name in de query string or body.

If you change the url into: http://localhost:7071/api/MySampleFunction?name=Jan you will see an Hello Jan message.

Press **Ctrl-C** to stop the Azure Functions host.

Now open the folder in Visual Studio code by typing: `code .`

Check the **Dockerfile** file:

```
# To enable ssh & remote debugging on app service change the base image to the one below
# FROM mcr.microsoft.com/azure-functions/node:3.0-appservice
FROM mcr.microsoft.com/azure-functions/node:3.0

ENV AzureWebJobsScriptRoot=/home/site/wwwroot \
    AzureFunctionsJobHost__Logging__Console__IsEnabled=true

COPY . /home/site/wwwroot

RUN cd /home/site/wwwroot && \
    npm install
```

You see that the Docker image uses a base image from Microsoft containing the Node.js runtime for Azure Functions. To see all base images for Azure Functions, see https://hub.docker.com/_/microsoft-azure-functions-base (this will only show the v2 images at this moment).

There seems to be an issue with used base image, so change the from into:

```
FROM mcr.microsoft.com/azure-functions/node:3.0-node12
```

Now create the Docker image by typing in the terminal window (can be started from within Visual Studio code using **Ctrl-`**):

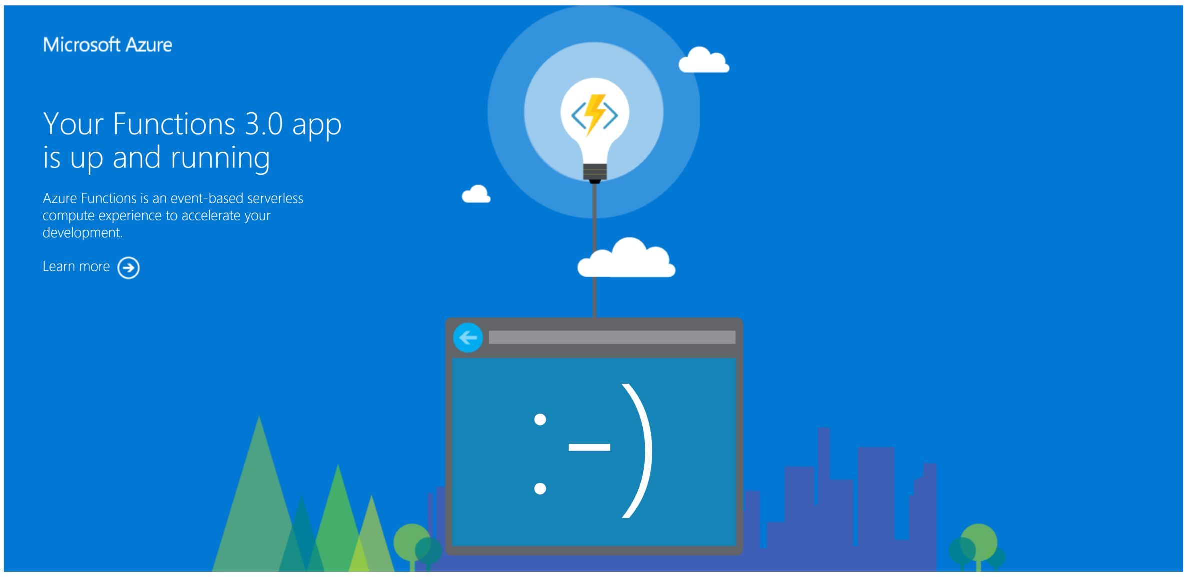
```
docker build -t myazurefuncimage .
```

If you type `docker image ls` you would see your new image in the list.

Test the image by running it:

```
docker run --name testfunction --rm -p 8080:80 myazurefuncimage
```

Open your browser and navigate to: http://localhost:8080/ to check if the function app is running:



Microsoft Azure

Your Functions 3.0 app is up and running

Azure Functions is an event-based serverless compute experience to accelerate your development.

Learn more

After starting the image you see a message how to invoke the function:

```
info: Microsoft.Azure.WebJobs.Script.WebHost.WebScriptHostHttpRoutesManager[0]
      Initializing function HTTP routes
      Mapped function route 'api/MySampleFunction' [get,post] to
'MySampleFunction'
```

However when trying http://localhost:8080/api/MySampleFunction?name=Jan you will notice that it is not possible to actually run the function. The reason is that by default Azure functions are secured by an Function Key which is only available after deploying to Azure. The local functions runtime we used earlier does not use this.

But we can make it working again.

Stop the Docker container by pressing **Ctrl-c**. Because of the `--rm` in the Docker command, the container will be automatically removed when stopped.

Open the file *function.json* and change `"authLevel": "function"` into `"authLevel": "anonymous"` and try again:

```
docker build -t myazurefuncimage .
docker run --name testfunction --rm -p 8080:80 myazurefuncimage
```

You will need to build image again to incorporate the change you made.

Before actually publishing the image, make sure you change the authLevel back to `"authLevel": "function"` .

Pushing the image to a container registry

To share your image, which includes deploying to Azure, you must push it to a registry, such as Docker Hub or Azure Container Registry.

Docker Hub

Open a command or terminal window and login into Docker Hub:

```
docker login
```

Tag your created Docker image:

```
docker tag myazurefuncimage:latest <docker_id>/myazurefuncimage:1.0
```

Change `<docker_id>` to your own Docker ID.

And push the image:

```
docker push <docker_id>/myazurefuncimage:1.0
```

Azure Container Registry

If you have an Azure Subscription and do not have an Azure Container registry yet, create one first either in the Azure Portal or by using Azure CLI:

```
az login
```

If you have multiple Azure directories under the same email account, the add the tenant (directory):

```
az login --tenant <tenant>.onmicrosoft.com
```

And create a new container registry:

```
az acr create --resource-group <resource-group-name> --location westeurope --sku
Basic --admin-enabled true --name <registry-name>
```

Tag your created Docker image:

```
docker tag myazurefuncimage:latest <registry-
name>.azurecr.io/myazurefuncimage:1.0
```

Change `<registry-name` to your Azure Container Registry name.

Login into the Azure Container registry using the admin credentials (can be found in the Azure Portal under Access Keys):

```
docker login <registry-name>.azurecr.io
```

And push the image:

```
docker push <registry-name>.azurecr.io/myazurefuncimage:1.0
```

Create Azure Function App

Next you will need to create a new Azure Function App which will use the pushed image as runtime stack.

Let's do it using Azure CLI.

When not yet logged in, type in a terminal window:

```
az login
```

Create a new resource group when you do no have one yet:

```
az group create --name rg-azurefunctionstest --location westeurope
```

> You can't host Linux and Windows apps in the same resource group. If you have an existing resource group named `rg-azurefunctionstest` with a Windows function app or web app, you must use a different resource group!

Create a general-purpose storage account in your resource group and region:

```
az storage account create --name <storage-name> --location westeurope --
resource-group rg-azurefunctionstest --sku Standard_LRS
```

> Replace `<storage-name` with a globally unique name appropriate to you. Names must contain three to 24 characters numbers and lowercase letters only.
>
> `Standard_LRS` specifies a typical general-purpose account

Nest create a Premium plan for Azure functions in the Elastic Premium 1 tier:

```
az functionapp plan create --resource-group rg-azurefunctionstest --name asp-
functions --location westeurope --number-of-workers 1 --sku EP1 --is-linux
```

Create the actual function app (replace `<storage-name>` and `<docker_id>` either by your DockerId or your Azure container registry name):

```
az functionapp create --name fa-dockertest001 --storage-account <storage_name> -
-resource-group rg-azurefunctionstest --plan asp-functions --deployment-
container-image-name <docker_id>/myazurefuncimage:1.0 --functions-version 3
```

> **The name of the function app is globally unique, so if you get an error while creating, use another name**.
>
> When using the Azure Container Registry you will get a message that you did not provide a credential. The command will try to look it up.

Copy the connection string of the created storage account to an environment variable:

In Bash:

```
connstring=$(az storage account show-connection-string --resource-group rg-
azurefunctionstest --name <storage_name> --query connectionString --output tsv)
```

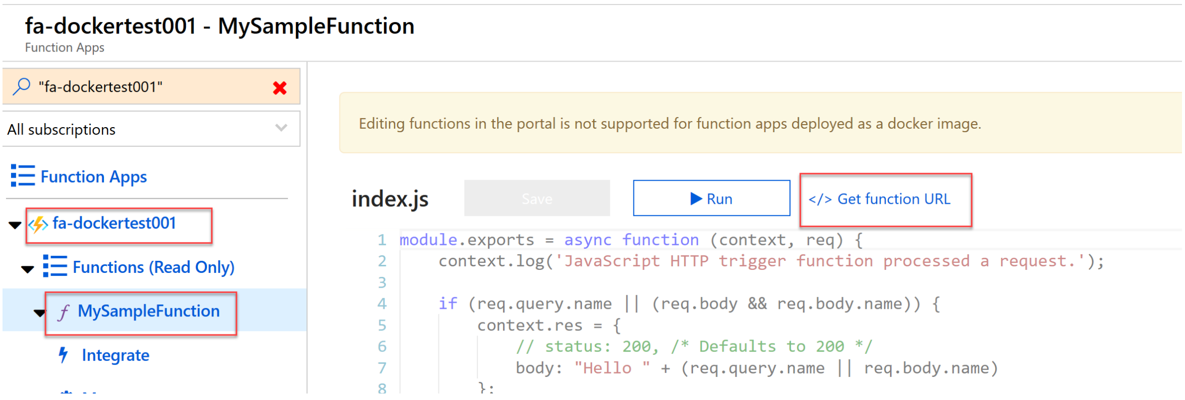
In Powershell:

```
$Env:connstring=$(az storage account show-connection-string --resource-group rg-
azurefunctionstest --name <storage_name> --query connectionString --output tsv)
```

Add the setting to the previously created functionapp:

```
az functionapp config appsettings set --name fa-dockertest001 --resource-group
rg-azurefunctionstest --settings AzureWebJobsStorage=${connstring}
```

In Azure Portal navigate to your created functionapp and function and click on Get function URL.


fa-dockertest001 - MySampleFunction
Function Apps

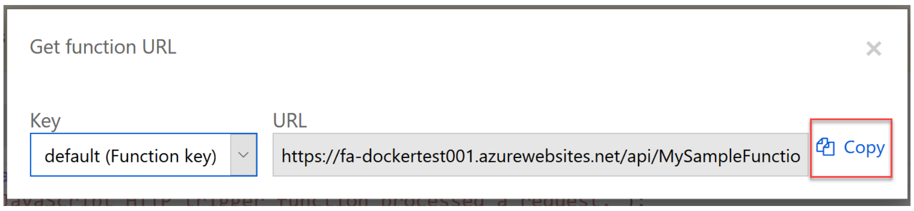
"fa-dockertest001"
All subscriptions
Function Apps
fa-dockertest001
Functions (Read Only)
MySampleFunction
Integrate

Editing functions in the portal is not supported for function apps deployed as a docker image.

index.js   Save   ▶ Run   </> Get function URL

```
module.exports = async function (context, req) {
    context.log('JavaScript HTTP trigger function processed a request.');

    if (req.query.name || (req.body && req.body.name)) {
        context.res = {
            // status: 200, /* Defaults to 200 */
            body: "Hello " + (req.query.name || req.body.name)
        };
```


Get function URL

Key: default (Function key)
URL: https://fa-dockertest001.azurewebsites.net/api/MySampleFunctio
Copy

And click on the Copy link.

Paste this a new browser tab and check that it works.

Now you can experiment by adding some changes to your code and pushing at as a new version to Docker Hub or Azure Container Registry:

```
docker build --tag <docker_id>/myazurefuncimage:1.0.1 .
docker push <docker_id>/myazurefuncimage:1.0.1
```

Then, because we did not created some continuous delivery pipeline, you have to update the functionapp image by hand either by using Azure Portal or Azure CLI.

In Azure CLI:

```
az functionapp config container set --name fa-dockertest001 --resource-group rg-
azurefunctionstest --docker-custom-image-name <docker_id>/myazurefuncimage:1.0.1
```

Connect to the container using SSH

With the above configuration you can not access the container from the Azure Portal using SSH (Kudo). You can use the Console bash but you have less privileges.

Update the container image in your Dockerfile to:

```
FROM mcr.microsoft.com/azure-functions/node:3.0-node12-appservice
```

Update the image again:

```
docker build --tag <docker_id>/myazurefuncimage:1.1 .
docker push <docker_id>/myazurefuncimage:1.1
```

And update the container image of the function app:

```
az functionapp config container set --name fa-dockertest001 --resource-group rg-
azurefunctionstest --docker-custom-image-name <docker_id>/myazurefuncimage:1.1
```

Now in the Azure Portal navigate to the Function App, select the Platform features tab and click on Advanced tools (Kudo).

A new tab will be opened.

Select SSH from the top navigation bar.

Type in `top` and you should see the top running processes, where `node` will be one of.

Note that you do not have an editor available. If you need that, you should add it to your Docker image. But it is not custom to do this, maybe only when the image is for development only.

Clean up resources

You can remove all the resources by removing the entire resource group:

```
az group delete --name rg-azurefunctionstest
```