Azure Container Registry – June ’17 Preview Features

|  |  |
| --- | --- |
| Role | Individuals |
| Owners | Steve Lasker |

Table of Contents

[Overview 2](#_Toc487110281)

[Key Takeaways 2](#_Toc487110282)

[Demo Bits and licenses 2](#_Toc487110283)

[Client Machine 2](#_Toc487110284)

[Docker For Windows 2](#_Toc487110285)

[Demo Reset 2](#_Toc487110286)

[Demo 1 – Create an Azure Container Registry 3](#_Toc487110287)

[Demo 2 – Web Hooks 3](#_Toc487110288)

[Demo 3 – Deletes 3](#_Toc487110289)

[project.json 3](#_Toc487110290)

* dd SQL Healthcheck to the containerized version

# Overview

## Key Takeaways

The three things viewers should take away from this section/demo will be:



# Demo Bits and licenses

## Client Machine

Windows 10 Client

Azure CLI 2.0 <https://docs.microsoft.com/en-us/cli/azure/overview>

## Docker For Windows

<https://download.docker.com/win/stable/InstallDocker.msi> Stable build

# Demo Reset

# Demo 1 – Create an Azure Container Registry

|  |  |  |
| --- | --- | --- |
| **Visual** | **Demo Steps** | **Talking Points** |
|  | <http://aka.ms/acr/portal/preview>  Click Container Registries | We’ll start by creating a new Azure Container Registry using the Azure Portal. We can create a registry using the Azure CLI as well, but it’s easier to show the options here in the portal |
|  | Create new registry | With the June preview, we now have some new options  We’ll start by providing a unique URL for our registry  We’ll create a resource group to keep our registry separate from other resources as we’ll likely use our registry for many different container hosting services  For Preview, we have a few regions, but we’ll roll this out to all regions as we go to general availability  You’ll notice we now have: Use managed registry:  This means we no longer leave the storage account in your subscription. This is also where all the new features light up including:   * Azure Active Directory for individual users * Web Hooks * Repository Delete   We’ll have more features coming as well. |
|  |  | We then have different SKUs  Basic is intended for individual developer scenarios. This is the cheapest option that enables all the features, but limits exist on scale and operations.  Standard is the SKU that supports most development teams, including larger amounts of storage for custom images and more operations such as Push and Pull  Premium will increase the storage and operations, but will also include some additional features that we haven’t yet rolled out, so stay tuned… |
|  | Press [Create] | We’ll let azure provision our new registry.  Now, you may have noticed we left the admin account disabled.  We no longer need to use the single admin account for all users, or configure a Service Principal for individual users.  We can now take advantage of your individual Azure Identity |
|  | Open Powershell  **az acr login -n [registry name]** | In a command window, we can login to our registry using: **az acr login -n [the name of our registry]**  This will take our azure identity, create an entry in the Windows Credentials Manager, and place the token in our docker config file for our registry |
|  | Open VS Code  Open %userprofile%/.docker  Open config.json | We can see the secure entry here for our registry, but notice all the credential are stashed aware securely in the Windows Credential Manager |
|  | Open Credential Manager  Select Windows Credentials  Scroll to find your registry | In the Windows Credential Manager, we can see our newly created registry |
|  | Powershell: | Now that we have our registry created and we’re individually authenticated, lets push an image to see how this all works |
|  | PS> Docker images | We’ll start with any image. I happen to have a web image I built with VS, but you can use any image on dockerhub |
|  | PS> docker tag web stevelasdemo.azurecr.io/demos/web:1  PS> docker push stevelasdemo.azurecr.io/demos/web:1 | By tagging our web image with our fully qualified registry name, we can simply push it to our registry |
|  | PS> az acr repository list -n stevelasdemo | We can see our images by calling the repository list CLI |
|  | Open the portal | We can also see our images using the Azure Portal |

# Demo 2 – Web Hooks

|  |  |  |
| --- | --- | --- |
| **Visual** | **Demo Steps** | **Talking Points** |
|  | Click Webhooks in the Registry Blade | In the cloud native workloads, it’s a common practice to stitch together a series of workflows through webhooks. Webhooks are really nice as we no longer need to poll a service, with a denial of service like attack to simply see if something changed.  It’s kinda like constantly checking your facebook page to see if anyone messaged you.  As opposed to getting a notification, a buzz, that says you’ve got a message. |
|  | Click **+ Add** | The Azure Container Registry now supports webhooks for your workflows. We’ll start by creating a new webhook  You’ll notice we’ll need a Service URI, which is the URI that will get called when an event happens.  We don’t yet have a URI, so lets walk backwards a bit to start with the end result |
|  |  | There are many things we can do with Webhooks. We might trigger a build of collection of images that are based on the image we just updated.  We might deploy, or likely trigger a series of tests  We might trigger vulnerability scanning, so these services no longer need to constantly ping for updates  For this demo, I’m going to post to Slack |
|  | Open Slack  Choose Settings, Add an app or integration | We’ll start by configuring slack to receive webhook notifications |
|  | Search **webhook** | We’ll find the incoming webhook integration |
|  | Add Configuration | And add the configuration |
|  | Click the green button | Here we can choose which channel we want for our notifications. If you have automated builds, you’ll likely want to post to specific channel, or you’ll flood your main channel |
|  | Copy the webhook url | As we scroll through, we can see we have quite a few options for how to configure Slack web hooks.  You’ll also notice we can configure how the message and icons are configured in the message.  This requires setting values on the body of the message posted  We’ll copy the URL from the bottom of the page with the copy link |
|  | Paste the Slack URI into Service URI  Click **[Create]** | If we go back to the Azure Portal, we can now configure the ACR Web Hook to make an outgoing call  We’ll give the webhook name a unique id.  This is a URI, so it needs to be lowercase, with no spaces.  We then provide the Service URI. This is the webhook we’ll call as actions are made to the registry |
|  | Click Ping | Once the webhook is created, we can easily test it with the Ping button |
|  | Click Refresh | We can see this didn’t actually work. We received a 500 error. |
|  | Click [Response] | If we look at the response, we can see there’s some sort of error related to: **missing\_text\_or\_fallback\_or\_attachments**  It turns out, Slack requires specific formatting of the body of the post.  While we support customizations of the headers, we don’t yet support customizations of the body. This is something we’ll iterate a bit during the preview.  However, it’s a great example of how we can stitch together other capabilities, such as Azure Functions to customize our post to Slack |

|  |  |  |
| --- | --- | --- |
|  | Add an Azure Function | We’ll create an Azure Function |
|  | Add a HTTP Trigger | We’ll use the HTTP Trigger template |
|  | Name: AcrToSlack  Paste in code from below | With our template in place, we’ll paste in the code we need  However, we’re using the NewtonSoft JSON Nuget, so we’ll just need to add that Nuget reference |
|  | Click View Files  Click Add project.json  Paste in the project.json configuration from below  [Save] | We’ll let functions know which nugets we need by creating a project.json file  As we save, we can see the Nuget restore complete |
|  | Open run.csx  Copy the Slack Webhook URL  Replace Uri \_uri = new Uri("**[WEBHOOK URL HERE]**"); | We’ll switch back to our code file and paste in the webhook URL |
|  | Click **[Save and run]** | With all the changes in place, lets give it a test run |
|  | Switch to Slack | We should see our posting to slack, based on some test data  Voila, nice. We at least have our test working from our Azure Function to Slack. |
|  | Click Get Function Key | Since we’re using a function to parse our ACR Notification, we just need to update ACR to notify our Azure Function, which will do the post to Slack |
|  | Pate into the ACR Service URI  **[Save]** | Now we just need to save the updated URI and give it a test |
|  | Click Ping | We can ping our Azure Function to test it from ACR |
|  | Switch to Slack | If we look in Slack, we can see our test |
|  | docker tag web stevelasdemo.azurecr.io/demos/web:2  docker push stevelasdemo.azurecr.io/demos/web:2 | We’ll create another tag and push it to the registry |
|  | Browse the portal webhooks | We can see a new notification we sent |
|  | Browse Azure Functions Logs | Looking at the Azure Function Logs, we can see it passed through |
|  | Browse Slack | And looking at Slack, we can see our notification came all the way through |

# Demo 3 – Deletes

|  |  |  |
| --- | --- | --- |
| **Visual** | **Demo Steps** | **Talking Points** |
|  | Click Configure on the ACR Webhook  Click Actions – select all | Now that we’re stuffing a bunch of images in our registries, you might want to delete specific images  But, we also want a notification  Lets reconfigure the webhook notifications to push delete notifications as well |
|  | az acr repository delete -n stevelastest --repository demos/web | Delete the image |

using System.Net;

using Newtonsoft.Json;

using System.Net.Http;

using System.Text;

public static async Task<HttpResponseMessage> Run(HttpRequestMessage req, TraceWriter log)

{

log.Info("ACR WebHook Posted");

// Get request body

dynamic data = await req.Content.ReadAsAsync<object>();

Uri \_uri = new Uri("https://hooks.slack.com/services/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");

string \_message = data?.ToString() ?? "test from azure function";

string \_channel = "#imageupdates";

string \_username = "webhookbot";

HttpClient \_httpClient = new HttpClient();

var payload = new

{

text = \_message,

\_channel,

\_username,

};

var serializedPayload = JsonConvert.SerializeObject(payload);

var response = \_httpClient.PostAsync(\_uri,

new StringContent(serializedPayload, Encoding.UTF8, "application/json")).Result;

return \_message == null

? req.CreateResponse(HttpStatusCode.BadRequest, response.ToString())

: req.CreateResponse(HttpStatusCode.OK, String.Format("Message: {0}, Posted to: {1}", \_message, \_channel));

}

## project.json

{

"frameworks": {

"net46":{

"dependencies": {

"Newtonsoft.Json": "10.0.3"

}

}

}

}