# Module 6 - Troubleshooting

This module covers the basics of troubleshooting issues with Web Apps on Linux

## Prerequisites

The following prerequisites are necessary before participating in this Boot Camp.

* Familiarity with the basic troubleshooting tools used for Azure App Service.
* Ability to write Kusto/Jarvis queries.
* Required Ramweb access for Kusto/Jarvis endpoints.

## Goal

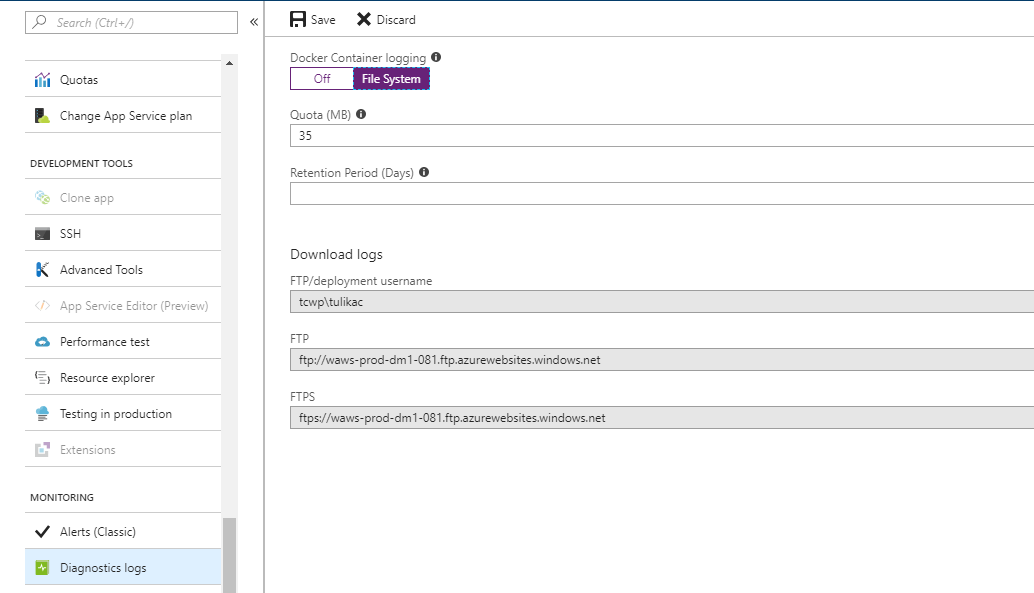
After completing this Boot Camp, you will have achieved a Level 200 skill set in:

* Logs available for troubleshooting Linux Web App issues.
* Identifying common troubleshooting scenarios.

# Docker Logs

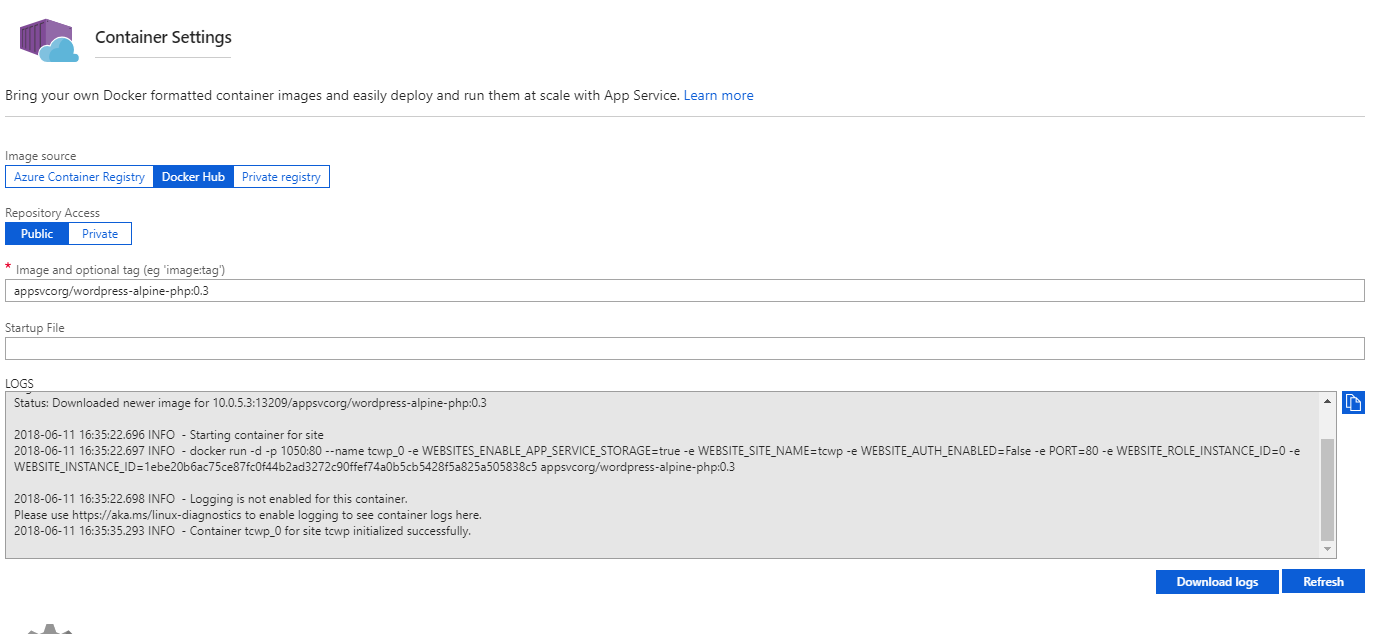
Docker logs are useful for troubleshooting issues related to container start. These logs show output from the Docker daemon and the STDOUT and STDERR information that is logged by a container.

By default, Docker logs are enabled for all our blessed images. However, if you want to get STDOUT and STDERR output from Web Apps for Containers, you'll need to enable logging. You can do that using the Diagnostic Logs blade as shown below.



You can view Docker logs using any one of these ways:

1. In the Container Settings blade as shown below.



(This is a truncated version of the logs, but you can download the full logs by clicking on Download Logs.)

1. FTP to the website and the logs will be located in the /Logfiles directory. You can also access this via the Kudu (Advanced Tools) Bash console. The naming convention for the Docker log is *YYYY\_MM\_DD\_RDxxxxxxxxxxxx\_docker.log*
2. Use the Kudu API that allows you to easily see the current Docker log details (such as the filename, etc.) and download the current Docker logs in Zip format.

To see details on the current Docker logs in JSON format, you can use this URL:

<https://[sitename].scm.azurewebsites.net/api/logs/docker>

You can get to this easily by going to Advanced Tools (Kudu) and then appending "/api/logs/docker" to the URL. The output of this will be a JSON response with the most relevant and current Docker logs.

|  |  |
| --- | --- |
| C:\Users\james\AppData\Local\Temp\SNAGHTML53be59c.PNG | It's a good idea to install a JSON viewer extension into your browser to get nicely formatted JSON output. |

If you want to download the logs shown in the above API in Zip format, append "zip" to the URL. For example:

<https://[sitename].scm.azurewebsites.net/api/logs/docker/zip>

|  |  |
| --- | --- |
| C:\Users\james\AppData\Local\Temp\SNAGHTML53be59c.PNG | You can also find a link to each of these APIs on the Kudu home page. |

A Docker log is created even if you don’t have logging enabled. However, you will only see output from the Docker daemon if logging is disabled. You won't see STDOUT and STDERR from the container. The example below is what you will see when logging is not enabled.

2018-06-15 12:23:43.487 INFO - Issuing docker pull: imagename =appsvcorg/wordpress-alpine-php:0.3

2018-06-15 12:23:44.406 INFO - docker pull returned STDOUT>> 0.3: Pulling from appsvcorg/wordpress-alpine-php

Digest: sha256:68a6f807771668f73d808a452f77462d1c8536cc1fb5776a9dcf404d1a673a1e

Status: Downloaded newer image for 10.0.5.2:13209/appsvcorg/wordpress-alpine-php:0.3

2018-06-15 12:23:44.529 INFO - Starting container for site

2018-06-15 12:23:44.529 INFO - docker run -d -p 11931:80 --name tcwp\_0 -e WEBSITES\_ENABLE\_APP\_SERVICE\_STORAGE=true -e WEBSITE\_SITE\_NAME=tcwp -e WEBSITE\_AUTH\_ENABLED=False -e PORT=80 -e WEBSITE\_ROLE\_INSTANCE\_ID=0 -e WEBSITE\_INSTANCE\_ID=1ebe20b6ac75ce87fc0f44b2ad3272c90ffef74a0b5cb5428f5a825a505838c5 appsvcorg/wordpress-alpine-php:0.3

2018-06-15 12:23:44.529 INFO - Logging is not enabled for this container.

Please use https://aka.ms/linux-diagnostics to enable logging to see container logs here.

In the snippet below, the important log entries related to container initialization are highlighted. This is an example of a successful container start. Also notice that there is much more information included in this output. That additional information was written by the container to the STDOUT and STDERR steams, and because logging is enabled, we can see that output in the Docker log.

2018-01-15 05:48:29.711 INFO - Issuing docker login to sever:

2018-01-15 05:48:32.695 INFO - docker login to succeeded

2018-01-15 05:48:32.702 INFO - Issuing docker pull isaocorp/mamoru-pay-app-server:staging

2018-01-15 05:48:54.652 INFO - docker pull returned STDOUT>> staging: Pulling from isaocorp/mamoru-pay-app-server

aa18ad1a0d33: Already exists

29d5f85af454: Already exists

eca642e7826b: Already exists

32b808b64451: Pull complete

bcf50932a8fa: Pull complete

Digest: sha256:a0b423a28081ba7fa1a52c93872616c4eab52ef0519a97a065ff51b97d5082b0

Status: Downloaded newer image for isaocorp/mamoru-pay-app-server:staging

2018-01-15 05:48:56.080 INFO - Starting container for site

2018-01-15 05:48:56.080 INFO - docker run -d -p 20801:80 --name mmr-pay-app-server\_\_e281\_0 -e WEBSITES\_ENABLE\_APP\_SERVICE\_STORAGE=false -e WEBSITE\_SITE\_NAME=mmr-pay-app-server -e WEBSITE\_AUTH\_ENABLED=False -e PORT=80 -e WEBSITE\_ROLE\_INSTANCE\_ID=0 -e WEBSITE\_INSTANCE\_ID=31fe78b60cd3475555aefb722fea9e6225bba3d491724d5ca88424ffcffa3908 -e HTTP\_LOGGING\_ENABLED=1 isaocorp/mamoru-pay-app-server:staging

2018-01-15 05:49:00.300 INFO - Container mmr-pay-app-server\_\_e281\_0 for site mmr-pay-app-server\_\_e281 initialized successfully.

2018-01-15 05:49:05.303 INFO - Container logs

2018-01-15T05:48:59.672573038Z AH00558: apache2: Could not reliably determine the server's fully qualified domain name, using 172.24.0.3. Set the 'ServerName' directive globally to suppress this message

2018-01-15T05:48:59.725344010Z AH00558: apache2: Could not reliably determine the server's fully qualified domain name, using 172.24.0.3. Set the 'ServerName' directive globally to suppress this message

2018-01-15T05:48:59.985434240Z [Mon Jan 15 05:48:59.982749 2018] [mpm\_prefork:notice] [pid 1] AH00163: Apache/2.4.10 (Debian) PHP/7.1.9 configured -- resuming normal operations

2018-01-15T05:48:59.985491539Z [Mon Jan 15 05:48:59.982811 2018] [core:notice] [pid 1] AH00094: Command line: 'apache2 -D FOREGROUND'

2018-01-15T05:49:00.301619383Z 172.24.0.1 - - [15/Jan/2018:05:48:59 +0000] "GET /robots933456.txt HTTP/1.1" 404 1717 "-" "-"

2018-01-15T05:49:00.465166309Z 172.24.0.1 - - [15/Jan/2018:05:49:00 +0000] "GET / HTTP/1.1" 200 974 "-" "ReadyForRequest/1.0 (LocalCache)"

2018-01-15T05:49:00.487880610Z 172.24.0.1 - - [15/Jan/2018:05:49:00 +0000] "HEAD / HTTP/1.1" 200 949 "-" "curl/7.47.0"

2018-01-15T05:49:00.609574371Z 172.24.0.1 - - [15/Jan/2018:05:49:00 +0000] "GET / HTTP/1.1" 200 972 "-" "ReadyForRequest/1.0 (AppInit)"

2018-01-15 05:51:00.305 INFO - Container logs

2018-01-15T05:50:55.586860663Z 172.24.0.1 - - [15/Jan/2018:05:50:55 +0000] "HEAD / HTTP/1.1" 200 949 "-" "curl/7.47.0"

## Startup Issues

One of the more common issues with container startup is timeout. When we start your container, we'll wait a while for it to start and initialize. We consider the startup to be successful once the container is running and once we get a response to a ping so that we know it's ready to respond to HTTP traffic. We'll wait 230 seconds for that to happen. If we don't have a successful start within 230 seconds, we'll assume there's a problem and we'll stop the container.

In the Docker logs, you will see something like this

2018-01-15 07:22:44.071 INFO - Issuing docker pull: imagename =appsvcorg/wordpress-alpine-php:0.1

2018-01-15 07:25:25.456 INFO - docker pull returned STDOUT>> 0.1: Pulling from appsvcorg/wordpress-alpine-php

1160f4abea84: Pulling fs layer

838fa68599a8: Pulling fs layer

…

ce3e5f94724a: Pull complete

e2644787df1b: Pull complete

Digest: sha256:9a546f7a7740ddcdb38f9a2ff82b7fc8128472eb3470c48a3c64ad68b11541d9

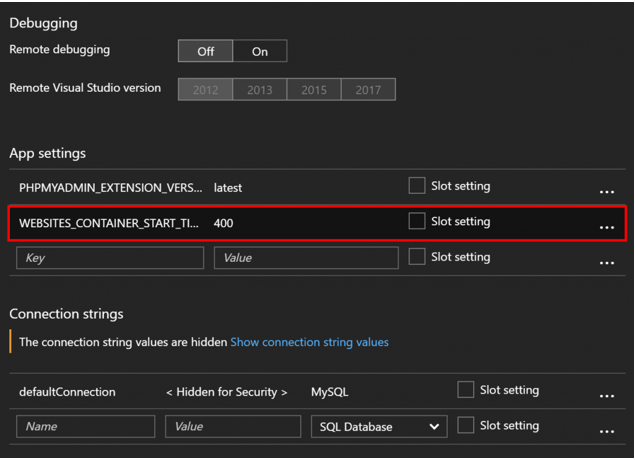
Status: Downloaded newer image for 10.0.0.24:13209/appsvcorg/wordpress-alpine-php:0.1

2018-01-15 07:25:25.696 INFO - Starting container for site

2018-01-15 07:25:25.699 INFO - docker run -d -p 1277:80 --name realidaddigitalwebapp\_0 -e WEBSITES\_ENABLE\_APP\_SERVICE\_STORAGE=true -e WEBSITE\_SITE\_NAME=realidaddigitalwebapp -e WEBSITE\_AUTH\_ENABLED=False -e PORT=80 -e WEBSITE\_ROLE\_INSTANCE\_ID=0 -e WEBSITE\_INSTANCE\_ID=61bcf85732cb9d2c995600e3db5b5c7283328572780b10c46059d466fb525c6d -e HTTP\_LOGGING\_ENABLED=1 appsvcorg/wordpress-alpine-php:0.1

2018-01-15 07:35:27.519 ERROR - Container realidaddigitalwebapp\_0 for site realidaddigitalwebapp did not start within expected time limit. Elapsed time = 600.4423138 sec

We realize that some containers might need more time to start, so we allow you to increase that 230 second wait time up to a limit of 1800 seconds. To configure that, add an app setting called **WEBSITES\_CONTAINER\_START\_TIME\_LIMIT** and set it to the number of seconds you would like for us to wait for your container to start.



Sometimes the container starts up successfully but doesn’t respond to a HTTP request and is considered not ready. Here's a snippet from a Docker log that illustrates an interesting problem.

2017-09-14 19:26:22.494 INFO - Starting container for site

2017-09-14 19:26:22.496 INFO - docker run -d -p 52879:80 --name customer-webapp . . .

2017-09-14 19:26:24.271 INFO - docker run returned: STDOUT>> c73ac7c0d7d2c1726d7a95929e591703d7c39eb7e7f314210ad5b4064f433762

. . .

2017-09-14T19:26:34.628607412Z { engine: 'joenode', port: '3000', pid: 5 }

2017-09-14 19:30:14.428 ERROR - Container customer-webapp\_0 for site customer-webapp did not start within expected time limit. Elapsed time = 230.0443067 sec>

We have truncated much of this output, but what you can see here is that the "docker run" command was run and two seconds later, you see a long, hexadecimal string as the return code. (This is on the third line of the log.) That hexadecimal string is the container's unique identifier, and the fact that we have a unique identifier for the container means that the container did start. However, at the end of the log, we see a message that the container did not start within the expected time limit.

|  |  |
| --- | --- |
| C:\Users\james\AppData\Local\Temp\SNAGHTML53be59c.PNG | FYI, we now use the Docker .NET SDK to run Docker commands and we no longer output the container ID. |

Since we know that the container started, and we also know that Web App for Containers doesn't consider the container to be started unless it's ready to respond to HTTP requests, we can conclude that the issue here is that the container is not responding to requests on port 3000. (Notice that the Node process is listening on port 3000 as shown in the next to the last line.) There are two solutions to this issue:

* Use the **EXPOSE** instruction in your Dockerfile to expose port 3000.
* Use the **WEBSITES\_PORT** app setting with a value of "3000" to expose that port.

# 

# Lab: Exploring Docker Logs

In this lab, you will create a Web App for Containers app that fails to launch. You will use the Docker logs to determine why the app doesn’t work.

## Task 1: Change PowerShell Execution Policy

By default, PowerShell does not allow for running scripts. You will need to change the PowerShell execution policy to run the deployment script in this lab.

1. Launch PowerShell as an Administrator.
2. At the PowerShell prompt, run the following command:

Set-ExecutionPolicy Unrestricted

1. At the prompt, type **Y** to change the execution policy. (When you finish this lab, it is recommended that you use the Set-ExecutionPolicy command to reset your execution policy to **Restricted**.)

## Task 2: Install the Azure PowerShell Module

The Azure PowerShell module allows you to interact with your Azure subscription in PowerShell.

1. If you closed PowerShell after Step 1, launch it again. (Make sure to run it as an Administrator.)
2. At the PowerShell prompt, enter the following command:

Install-Module -Name AzureRM

1. Answer **Y** to all prompts. Module installation may take a few minutes to complete.

## Task 3: Download the ARM Template and Deployment Script

The ARM template and deployment script for this lab are in GitHub.

1. Browse to <https://github.com/jamesche75/LinuxBootCamp>.
2. Click the green **Clone or Download** button.
3. Click **Download ZIP** to download a Zip file containing the template and script.
4. Extract the Zip file to your Desktop or another convenient location.

## Task 4: Edit the Parameters File

The ARM template will automatically create an App Service Plan called BootCampPlan. However, you will need to edit the parameters file and specify a unique name for your Web App.

1. Open the file *parameters.json* in a text editor.
2. Edit the value for the *sites\_Module6\_name* parameter and change it from **YourSiteName**to a unique name for your Web App. (I recommend you use a name followed by your initials so that you can guarantee it’s unique.)
3. Save *parameters.json*.

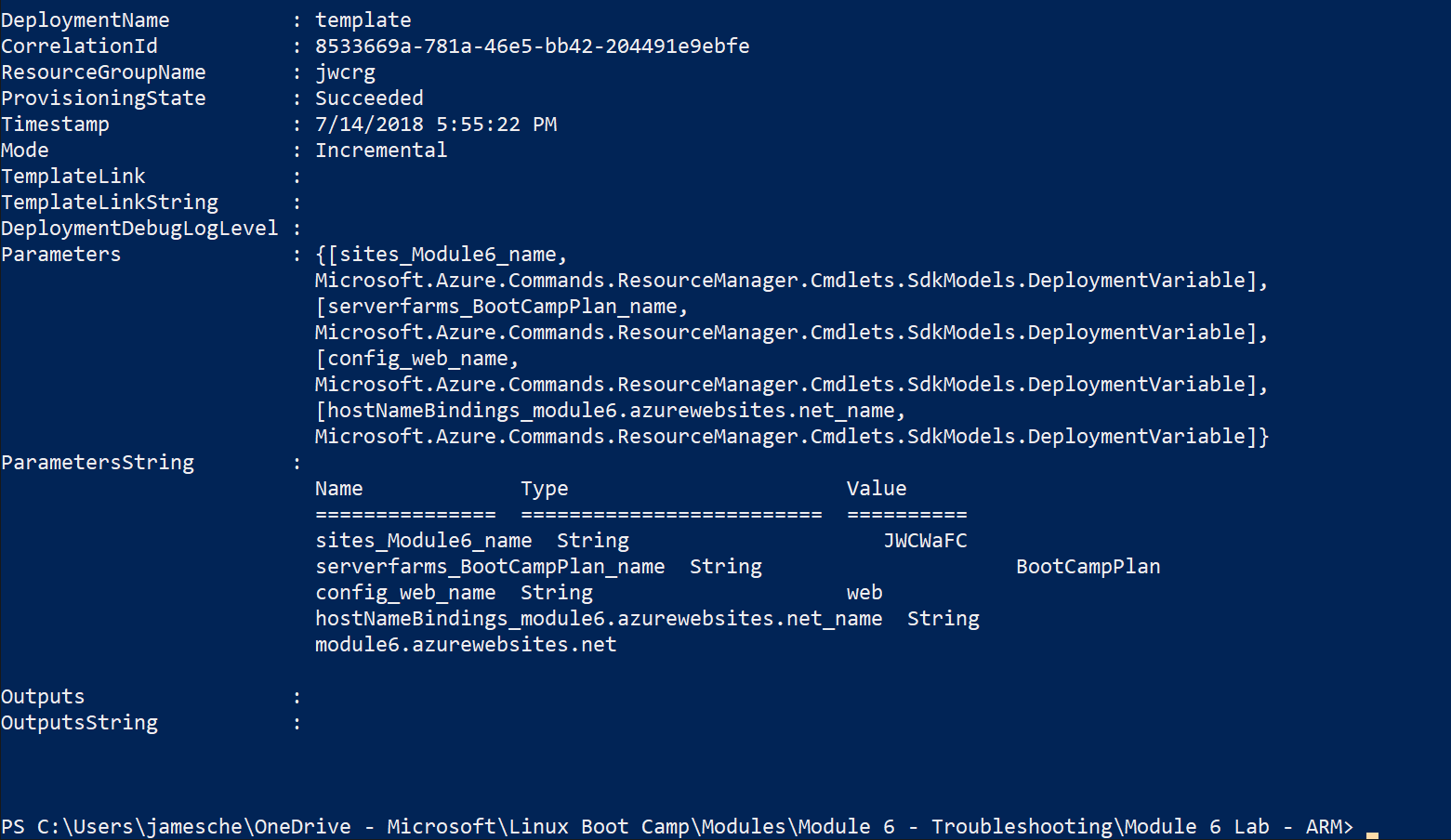
## Task 5: Run the Deployment Script

The Zip file you just downloaded contains an ARM template and a deployment script you can use to deploy the Web App for Containers app to your Azure subscription.

1. In PowerShell, change into the folder where you extracted the Zip file in Task 3.
2. Run the following command:

./deploy.ps1

1. Enter **R** at the prompt to run the script.
2. Enter your subscription ID when prompted.
3. Enter a unique name for the resource group. Make sure the name has no spaces. (The script will create this resource group if it doesn’t already exist.)
4. Enter a deployment name. (This can be anything you want to use. For example, **Test Deploy**.)
5. When prompted, log into your Azure account.
6. If the resource group you entered in Step 5 doesn’t already exist, enter a location where the new resource group will be created. (For example, **West US** or **North Europe**.)
7. Wait for deployment to succeed. When it succeeds, you will see output like the following:



## Task 6: Test the App

Browse to your app and figure out why it doesn’t work.

1. Browse to your app’s URL.
2. Wait for *Service Unavailable* in the browser. This indicates the Docker container did not start.
3. Observe the Docker logs to see what went wrong.
4. Enable Docker Logging in the Diagnostics Logs blade to see more information.

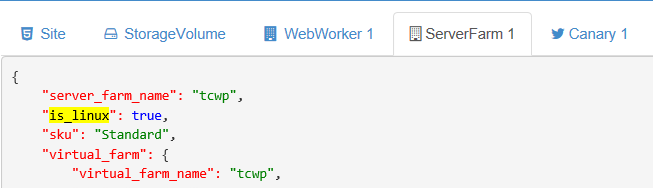
Do not delete the Web App at this point. Once you learn how to query Kusto for Linux issues, you can use this app as a test app for Kusto queries.

# Using Observer for Web Apps on Linux

You query a Linux site in the same way you query a Windows-based site. Here are some of the properties that you can check.

## Verifying the Linux OS

Go to the Server Farm tab and check the “is\_linux” property. If it's "true", the site is running on Linux.



## Determining Docker Configuration

You can use Observer to see if an app is running App Service on Linux, Web App for Containers with a single container, or Web App for Containers with a multi-container deployment.

Go to the Site tab and check the “linux\_fx\_version” property.

* App Service on Linux will show the runtime stack and version. For example:

"linux\_fx\_version": "php|7.0"

* Web App for Containers with a single container will start with "DOCKER" and will show the path to the container. For example:

"linux\_fx\_version": "DOCKER|appsvcorg/wordpress-alpine-php:0.3"

* Multi-container deployments will display a Base64 encoded string. The example below shows a Docker Compose multi-container deployment.

"linux\_fx\_version": "COMPOSE|",

Note that since the string is Base64 encoded, you can decode it easily. For example, if we decode the above string, this is what we get.

version: '3.3'

services:

db:

image: mysql:5.7

volumes:

- db\_data:/var/lib/mysql

restart: always

environment:

MYSQL\_ROOT\_PASSWORD: <root password>

MYSQL\_DATABASE: wordpress

MYSQL\_USER: <username>

MYSQL\_PASSWORD: <password>

wordpress:

depends\_on:

- db

image: wordpress:latest

ports:

- "8000:80"

restart: always

environment:

WORDPRESS\_DB\_HOST: db:3306

WORDPRESS\_DB\_USER: <username>

WORDPRESS\_DB\_PASSWORD: <password>

volumes:

db\_data:

## Instance Tenant ID and Name

The WebWorker tab shows you the workers the site is currently running on, the instance name, and the tenant ID. Note that all Linux stamps are megastamps, so you need the tenant IDs to correctly identify a worker. Different workers can be on different tenants.



The *stamp\_deployment\_id* property is the tenant ID.

# Troubleshooting Issues using Kusto

Here is a list of tables which are related to Web Apps for Linux.

|  |  |
| --- | --- |
| **Table name** | **Description** |
| LinuxRoleInstanceHeartBeats | As the name suggests, this is a HeartBeat table which has an entry every 5 mins for every instance |
| AntaresIISLogFrontEndTable | IIS requests which reached the Frontend. The Frontend for Linux and Windows stamps are the same |
| WorkerApacheAccessLogEventTable | This table logs the requests that came to the nginx server. It is very similar in structure to the AntaresIISLogWorkerTable |
| WorkerApacheErrorsEventTable | This is an error log for nginx. It is useful for debugging cold starts and request timeouts |
| WorkerSystemStats | This table logs the output of the following commands run on the instance every 5 mins   * df * top * iotop * iostat * ps * docker\_ps * cpu\_idle * cpu\_user * cpu\_system |
| LinuxRuntimeEvents | This table has various information regarding site container start/stop/recycle, including reasons why the container gets recycled (e.g. storage volume failover). This table also has LWAS restart events. |
| LinuxSiteStats | This table contains CPU and memory percentage per site per instance. We can use this table for knowing worker instance allocations as well. |
| WorkerSyslogEventTable | This table is logging basic Linux system information. It also logs docker daemon events |
| DefaultLogEventTable | It contains Linux kern.log. It includes information about RolePatcher, MSDS, Authpriv, Cron, etc |
| WorkerAzureAgentEventTable | Guest agent logs |
| WorkerOnStartEventTable | Outputs boot start phase on onstart |

# Basics of Troubleshooting using Kusto

If a customer complains about a website not being available or reporting errors, we start troubleshooting by looking at the Frontend table.

AntaresIISLogFrontEndTable

| where PreciseTimeStamp >= datetime(2018-06-17 10:25:00) and PreciseTimeStamp <= datetime(2018-06-17 10:40:00)

| where Cs\_host =~ "tctomcatimg.azurewebsites.net"

| where User\_agent != "AlwaysOn"

| project PreciseTimeStamp, Cs\_uri\_stem, Sc\_status, Sc\_substatus, Sc\_win32\_status, Time\_taken, WorkerHttpStatus, ServerRouted, SourceMoniker

| order by PreciseTimeStamp asc

## 

This shows you not just the status of the request, but also the IP address of the worker which served the request. You can take this IP address and query the LinuxRoleInstanceHeartbeats table to find out the name of the **r**ole instance.

LinuxRoleInstanceHeartBeats

| where PreciseTimeStamp >= datetime(2018-06-16 09:20:00) and PreciseTimeStamp <= datetime(2018-06-17 09:20:00)

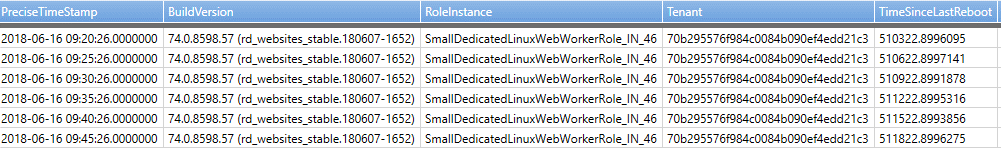
| where IpAddress == "10.0.4.23" and SourceMoniker == "LWAWSPRODDM1081"

| order by PreciseTimeStamp asc

| project PreciseTimeStamp, BuildVersion, RoleInstance, Tenant, TimeSinceLastReboot

Notice that we add the letter “L” to the SourceMoniker since this is a Linux stamp

You can now get the RoleInstance and the **T**enant on which the web app was running. You can also determine the build this instance is running and if it has been rebooted during that timeframe.



After getting the RoleInstance and Tenant, most of the following queries use those values instead of the web app name.

You can check the Nginx logs to verify if the request made it to the worker.

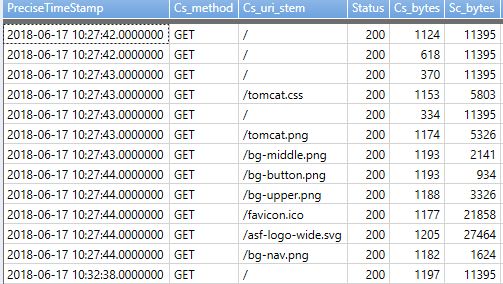
WorkerApacheAccessLogEventTable

| where PreciseTimeStamp >= datetime(2018-06-17 10:20:00) and PreciseTimeStamp <= datetime(2018-06-17 10:40:00)

| where RoleInstance == "SmallDedicatedLinuxWebWorkerRole\_IN\_46" and Tenant == "70b295576f984c0084b090ef4edd21c3" and Sitename =~ "tctomcatimg"

| order by PreciseTimeStamp asc

| project PreciseTimeStamp, Cs\_method, Cs\_uri\_stem, Status, Cs\_bytes, Sc\_bytes



If you want to look at the container details, you can look at the LWAS table. However, keep in mind that the LWAS table is very verbose, so you need to be careful about what information you are looking for.

WorkerLWASEventTable

| where PreciseTimeStamp >= datetime(2018-06-17 10:20:00) and PreciseTimeStamp <= datetime(2018-06-17 10:40:00)

| where RoleInstance == "SmallDedicatedLinuxWebWorkerRole\_IN\_46" and Tenant == "70b295576f984c0084b090ef4edd21c3" //and Msg contains "StartHostName"

| project PreciseTimeStamp, Facility, Msg

| order by PreciseTimeStamp asc

If this is a web app that is starting, you should see some statements like this:

PreciseTimeStamp Facility Msg

2018-06-17 10:24:33.0000000 tctomcatimg [11]: Issuing StartHostName: site: tctomcatimg, host: tctomcatimg.azurewebsites.net

2018-06-17 10:24:34.0000000 tctomcatimg [214]: Issuing start command to container: tctomcatimg\_0

You might also see exceptions like this:

[214]: image inspect for library/tomcat:latest failed ex = System.AggregateException: One or more errors occurred. ---> Docker.DotNet.DockerImageNotFoundException: Docker API responded with status code=NotFound, response={"message":"no such image: library/tomcat:latest: No such image: tomcat:latest"}

In general, you can ignore these exceptions. This exception is caused by the Docker daemon searching for the image on the local disk or the Container Registry.

You will see a message for the image being pulled, the container created, the docker run command, waiting for initialization and a final message to indicate that the container has been successfully started.

PreciseTimeStamp Facility Msg

2018-06-17 10:24:34.0000000 tctomcatimg [214]: Pulling image: tomcat:latest

2018-06-17 10:24:35.0000000 tctomcatimg\_0 [214]: Pulling from docker hub directly...

….

2018-06-17 10:27:17.0000000 tctomcatimg [214]: Issuing start command for site to docker daemon: docker run -d -p 58994:8080 --name tctomcatimg\_0 -e WEBSITE\_SITE\_NAME=tctomcatimg -e WEBSITE\_AUTH\_ENABLED=False -e PORT=8080 -e WEBSITE\_ROLE\_INSTANCE\_ID=0 -e WEBSITE\_INSTANCE\_ID=1ebe20b6ac75ce87fc0f44b2ad3272c90ffef74a0b5cb5428f5a825a505838c5 tomcat:latest

2018-06-17 10:27:17.0000000 DOCKERCLIENT [214]: Container created successfully: Id = 0e8c16e71a5874f46ac7cdb73bd734f2f3a4102d50f2cb6ae4646d8c05db5798, Warnings =

2018-06-17 10:27:18.0000000 DOCKERCLIENT [214]: Container started successfully: Id = 0e8c16e71a5874f46ac7cdb73bd734f2f3a4102d50f2cb6ae4646d8c05db579

2018-06-17 10:27:18.0000000 tctomcatimg [214]: Exposed port: 8080

2018-06-17 10:27:18.0000000 tctomcatimg [214]: Created container tctomcatimg\_0 for site tctomcatimg

2018-06-17 10:27:19.0000000 tctomcatimg [214]: Waiting for container ready: tctomcatimg\_0

2018-06-17 10:27:19.0000000 tctomcatimg [214]: IsHttpEndpointReady returned false for http://127.0.0.1:58994/robots933456.txt, : [AggregateEx, Type = System.Net.Http.HttpRequestException, Message = An error occurred while sending the request]

2018-06-17 10:27:35.0000000 tctomcatimg [214]: StartContainerUnit successful: tctomcatimg\_0

You will see similar kinds of messages for the Kudu container as well.

## You will also see logs with the facility called LWAS\_STATE. You can use these entries to determine the websites running on this instance.

## 214]: Sites >>>

## Site: mawscanary-750f6b19-9efd0005, State: Started

## ActiveContainer: ContainerName = mawscanary-750f6b19-9efd0005\_0, ImageName = appsvc/linuxcanary-runtime CreateTime = 6/12/2018 11:39:57 PM, LastKnownState-IsRunning: True, LastHealthCheckTime: 6/17/2018 10:39:45 AM SiteHome: /var/LWASFiles/Sites/mawscanary-750f6b19-9efd0005/home

## StorageVolumes: Primary = LocalPath = /var/LWASFiles/Sites/mawscanary-750f6b19-9efd0005/home, RemotePath = //10.0.176.8/volume-25-default/..., State = Healthy, LastHealthCheckTime 6/...

## Standby = LocalPath = /var/LWASFiles/Sites/mawscanary-750f6b19-9efd0005/ro\_home, RemotePath = //10.0.176.23/volume-25-default-standby/..., State = NotInitialized, LastHealthCheckTime 1/...

## SiteStorageVolumeManager-LastHealthCheckTime = 6/17/2018 10:39:45 AM

## Site: ~1tctomcatimg, State: Started

## ActiveContainer: ContainerName = tctomcatimg\_Kudu\_0, ImageName = appsvc/kudu:1805220018 CreateTime = 6/17/2018 10:24:29 AM, LastKnownState-IsRunning: True, LastHealthCheckTime: 6/17/2018 10:39:43 AM SiteHome: /var/LWASFiles/Sites/~1tctomcatimg/home

## StorageVolumes: Primary = LocalPath = /var/LWASFiles/Sites/~1tctomcatimg/home, RemotePath = //10.0.176.8/volume-25-default/..., State = Healthy, LastHealthCheckTime 6/...

## Standby = LocalPath = /var/LWASFiles/Sites/~1tctomcatimg/ro\_home, RemotePath = //10.0.176.23/volume-25-default-standby/..., State = NotInitialized, LastHealthCheckTime 1/...

## SiteStorageVolumeManager-LastHealthCheckTime = 6/17/2018 10:39:42 AM

## Site: tctomcatimg, State: Started

## ActiveContainer: ContainerName = tctomcatimg\_0, ImageName = tomcat:latest CreateTime = 6/17/2018 10:27:18 AM, LastKnownState-IsRunning: True, LastHealthCheckTime: 6/17/2018 10:39:45 AM SiteHome: /var/LWASFiles/Sites/tctomcatimg/home

## StorageVolumes: Primary = LocalPath = /var/LWASFiles/Sites/tctomcatimg/home, RemotePath = //10.0.176.8/volume-25-default/..., State = Healthy, LastHealthCheckTime 6/...

## Standby = LocalPath = /var/LWASFiles/Sites/tctomcatimg/ro\_home, RemotePath = //10.0.176.23/volume-25-default-standby/..., State = NotInitialized, LastHealthCheckTime 1/...

## SiteStorageVolumeManager-LastHealthCheckTime = 6/17/2018 10:39:44 AM

## Networks >>>

## Name: tctomcatimg\_nw, SiteName: tctomcatimg, CreateTime: 6/17/2018 10:24:26 AM IsRefCounted; False CurrentRefCount: 0

## Name: mawscanary-750f6b19-9efd0005\_nw, SiteName: mawscanary-750f6b19-9efd0005, CreateTime: 6/12/2018 11:39:54 PM IsRefCounted; False CurrentRefCount: 0

## Another way of determining which containers are running is by checking the WorkerSystemStats table.

WorkerSystemStats

| where PreciseTimeStamp >= datetime(2018-06-17 10:20:00) and PreciseTimeStamp <= datetime(2018-06-17 10:40:00)

| where RoleInstance == "SmallDedicatedLinuxWebWorkerRole\_IN\_46" and Tenant == "70b295576f984c0084b090ef4edd21c3" and Facility == "docker\_ps"

| order by PreciseTimeStamp asc

| project PreciseTimeStamp, Msg

## CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES SIZE

## 0e8c16e71a5874f46ac7cdb73bd734f2f3a4102d50f2cb6ae4646d8c05db5798 tomcat:latest "catalina.sh run" 7 minutes ago Up 7 minutes 127.0.0.1:58994->8080/tcp tctomcatimg\_0 89.6kB (virtual 467MB)

## 050fb2b157731623f30a2d6b778bca2436378df8a29ab38f4d54c35593a65da9 appsvc/kudu:1805220018 "/startup.sh 1009 f6303d48e5647c4fe17b11b3 1005 f6303d48e5647c4fe17b11b3 tctomcatimg" 10 minutes ago Up 10 minutes 127.0.0.1:56875->8181/tcp tctomcatimg\_Kudu\_0 25.8MB (virtual 4.84GB)

## 690ffa9c944ea90f7da6d404c9ad063b0629eba8067b78b8ed90ef9b811fc5af appsvc/linuxcanary-runtime "node /opt/src/canary.js" 4 days ago Up 4 days 127.0.0.1:15121->3000/tcp mawscanary-750f6b19-9efd0005\_0 0B (virtual 84MB)

## Lab: Looking at Kusto Tables

Create a custom Docker container by using the image tomcat:latest and determine container start by following the commands above.

## Identifying Container Startup Issues

You can start looking at the LWAS event table. I have reproduced a small snippet of the logs.

TIMESTAMP Facility Msg

2018-06-16 14:09:00.0000000 cbthambotlinux [924]: Issuing StartHostName: site: cbthambotlinux, host: cbthambotlinux.azurewebsites.net

2018-06-16 14:09:00.0000000 cbthambotlinux [924]: Attempting to provision the site.:

..

2018-06-16 14:09:00.0000000 cbthambotlinux [1110]: Issuing start command for site to docker daemon: docker run -d -p 40723:8080 --name cbthambotlinux\_0 -e WEBSITE\_NODE\_DEFAULT\_VERSION=8.1 -e APPSETTING\_WEBSITE\_NODE\_DEFAULT\_VERSION=8.1 -e WEBSITE\_SITE\_NAME=cbthambotlinux -e WEBSITE\_AUTH\_ENABLED=False -e WEBSITE\_ROLE\_INSTANCE\_ID=0 -e WEBSITE\_INSTANCE\_ID=9ec226f3bb3da09bee3562ac267eed0bbafce2c259ba5dbcdfb52a0b39f9eb85 appsvc/node:8.1.4\_1804040228 pm2 start app.js --no-daemon

2018-06-16 14:09:00.0000000 DOCKERCLIENT [1110]: Container created successfully: Id = 6445598bab7a40be2988d8a1a3fad5277c61f942b08f9147dc114dda0c64f466, Warnings =

2018-06-16 14:09:00.0000000 cbthambotlinux [1110]: Created container cbthambotlinux\_0 for site cbthambotlinux

2018-06-16 14:09:00.0000000 DOCKERCLIENT [1110]: Container started successfully: Id = 6445598bab7a40be2988d8a1a3fad5277c61f942b08f9147dc114dda0c64f466

2018-06-16 14:09:00.0000000 cbthambotlinux [1110] Selecting image for node 8.1

2018-06-16 14:09:00.0000000 cbthambotlinux [1110]: IsHttpEndpointReady returned false for http://127.0.0.1:40723/robots933456.txt, : [AggregateEx, Type = System.Net.Http.HttpRequestException, Message = An error occurred while sending the request]

2018-06-16 14:09:00.0000000 cbthambotlinux [1110]: Waiting for container ready: cbthambotlinux\_0

..

2018-06-16 14:13:00.0000000 cbthambotlinux [1110]: Start failed for LinuxSite with ex: System.Exception: docker container could not be started: cbthambotlinux\_0

at LWAS.LinuxSite.HandleNonStartedContainer (LWAS.DockerContainer dockerContainer) [0x0003d] in <009410c40f8c44de964dcb62a47b59af>:0

at LWAS.LinuxSite.StartContainer (LWAS.LinuxSiteStartInfo startInfo, System.String siteHome, System.Int32 port, LWAS.DockerNetwork network) [0x0016e] in <009410c40f8c44de964dcb62a47b59af>:0 currentState: Starting

2018-06-16 14:13:00.0000000 cbthambotlinux [928]: Issuing stop command to container: cbthambotlinux\_0

2018-06-16 14:13:00.0000000 cbthambotlinux [928]: Executing stop async for container: cbthambotlinux\_0

2018-06-16 14:13:00.0000000 cbthambotlinux [1110]: Issuing stop async for container: cbthambotlinux\_0

2018-06-16 14:13:00.0000000 cbthambotlinux [1110]: Container cbthambotlinux\_0 couldn't be started: Logs = 2018-06-16T14:09:44.876698404Z \_\_\_\_\_

2018-06-16 14:13:00.0000000 cbthambotlinux [1110]: Try get container logs for: cbthambotlinux\_0

2018-06-16 14:13:00.0000000 cbthambotlinux [1110]: Container cbthambotlinux\_0 did not start within the expected time limit. Elapsed time = 233.096504

Notice how the container did start but did not respond to a HTTP request and finally timed out. We should ask the customer to increase the value of WEBSITES\_CONTAINER\_START\_TIME\_LIMIT and see if it makes a difference.

## Identifying Storage Failovers

You can identify storage failovers by looking at LWAS table. For example:

WorkerLWASEventTable

| where PreciseTimeStamp >= datetime(2018-06-06 07:00:00) and PreciseTimeStamp <= datetime(2018-06-06 12:00:00)

| where RoleInstance == "SmallDedicatedLinuxWebWorkerRole\_IN\_68" and Tenant == "4be4b0bdc08e41d0b7faa2da5187ff6d"

| project PreciseTimeStamp, Facility, Msg

| order by PreciseTimeStamp asc

You will see messages like:

PreciseTimeStamp Facility Msg

2018-06-06 07:13:58.0000000 mawscanary-a1215bcb-03d76345 [21919]: StorageVolumeCheck failed before timeout - incrementing failedPings by 3 so we use readonly soon

2018-06-06 07:13:58.0000000 TestFileShareReadOperation [24519]: ls returned exit code: 2, stdout: , stdErr: ls: cannot access '/var/LWASFiles/Sites/pscssamlservice/home/LogFiles': Input/output error

2018-06-06 07:13:58.0000000 pscssamlservice [24519]: Storage volume test Path= /var/LWASFiles/Sites/pscssamlservice/home, Result= FAILURE, Reason = file access failure.

2018-06-06 07:13:58.0000000 pscssamlservice [24519]: StorageVolumeCheck failed before timeout - incrementing failedPings by 3 so we use readonly soon

Once a storage failover happens, the container will be restarted and it will point to a read-only volume. However, a container will be affected by storage failovers only if they have the app setting WEBSITES\_ENABLE\_APP\_SERVICE\_STORAGE set to True. If this setting is False, the container is not affected and will not restart.

|  |  |
| --- | --- |
| C:\Users\james\AppData\Local\Temp\SNAGHTML53be59c.PNG | If a storage volume changes, it causes a non-overlapped recycle. |

## Identifying Application Restarts

This information is also logged in the LWAS table. For example:

WorkerLWASEventTable

| where PreciseTimeStamp >= datetime(2018-06-06 06:00:00) and PreciseTimeStamp <= datetime(2018-06-06 12:00:00)

| where RoleInstance == "SmallDedicatedLinuxWebWorkerRole\_IN\_68" and Tenant == "4be4b0bdc08e41d0b7faa2da5187ff6d"

| project PreciseTimeStamp, Facility, Msg

| order by PreciseTimeStamp asc

PreciseTimeStamp Facility Msg

2018-06-06 09:07:33.0000000 [21956]: Attempting to start container for framework: PHP version: 7.2

2018-06-06 09:07:33.0000000 [21956]: Assigned port number: 9217

2018-06-06 09:07:33.0000000 [21956]: GetRandomPort

2018-06-06 09:07:33.0000000 pscssamlservice [24519]: Fetching startsiteContextForSite for pscssamlservice isMainSite: True

2018-06-06 09:07:33.0000000 pscssamlservice [24519]: Fetched token from cache

2018-06-06 09:07:33.0000000 pscssamlservice [24519]: SiteChanged: pscssamlservice IsMainSite: True

2018-06-06 09:07:33.0000000 pscssamlservice [24519]: Main Site changed : pscssamlservice, LastModifiedTime: 6/6/2018 9:07:29 AM, CheckPoint: 6/6/2018 7:14:03 AM

2018-06-06 09:07:33.0000000 pscssamlservice [24519]: ChangeNotificationHandler called ... pscssamlservice

2018-06-06 09:07:33.0000000 [21956]: Change notification processed, mustRecycle = True, primaryChanged = False, standbyChanged = False

2018-06-06 09:07:33.0000000 [21956]: Recycling container because of AppSettingsChange and isMainSite = False

Another way to identify recycles is to look in the LinuxRuntimeEvents table.

Run: [[Kusto.Explorer](https://wawseas.kusto.windows.net:443/wawsprod?query=H4sIAAAAAAAEAEWMwQqCQBRF90H%2f8HBVMAv9gAIJAyEjJn9gGG%2f1St%2bUPish%2bvawjcvDPffsWPq37UW5QfaEaDeffeh1QQsq8yI7lmlxoDW5c1gkl%2bU0%2fuW9a0CrL0WbIOpY0Fr4wdewePToFFVE40XdDZTEcTzCvQ1XeJ3yhrbOc806mClryMJ1QQylVcXKQVydyykQ%2fQBjmIOyswAAAA%3d%3d)] [[Kusto.Explorer on SAW](https://wawseas.kusto.windows.net:443/wawsprod?query=H4sIAAAAAAAEAEWMwQqCQBRF90H%2f8HBVMAv9gAIJAyEjJn9gGG%2f1St%2bUPish%2bvawjcvDPffsWPq37UW5QfaEaDeffeh1QQsq8yI7lmlxoDW5c1gkl%2bU0%2fuW9a0CrL0WbIOpY0Fr4wdewePToFFVE40XdDZTEcTzCvQ1XeJ3yhrbOc806mClryMJ1QQylVcXKQVydyykQ%2fQBjmIOyswAAAA%3d%3d&saw=1)] [[Kusto.WebExplorer](https://web.kusto.windows.net/clusters/wawseas.kusto.windows.net/databases/wawsprod?q=H4sIAAAAAAAEAEWMwQqCQBRF90H%2f8HBVMAv9gAIJAyEjJn9gGG%2f1St%2bUPish%2bvawjcvDPffsWPq37UW5QfaEaDeffeh1QQsq8yI7lmlxoDW5c1gkl%2bU0%2fuW9a0CrL0WbIOpY0Fr4wdewePToFFVE40XdDZTEcTzCvQ1XeJ3yhrbOc806mClryMJ1QQylVcXKQVydyykQ%2fQBjmIOyswAAAA%3d%3d)] [[Lens](https://lens.msftcloudes.com/v2/#/discover/query//results?datasource=(cluster:wawseas.kusto.windows.net,database:wawsprod,type:Kusto)&query=H4sIAAAAAAAEAEWMwQqCQBRF90H%2f8HBVMAv9gAIJAyEjJn9gGG%2f1St%2bUPish%2bvawjcvDPffsWPq37UW5QfaEaDeffeh1QQsq8yI7lmlxoDW5c1gkl%2bU0%2fuW9a0CrL0WbIOpY0Fr4)] <https://wawseas.kusto.windows.net:443/wawsprod>

LinuxRuntimeEvents

| where TIMESTAMP > ago(1h)

| where EventName =~ "ContainerRecycleRequested"

| take 1000

| project TIMESTAMP, Facility, EventName, Reason, AdditionalInfo

Facility EventName Reason AdditionalInfo

mawscanary-57158a59-e8caf9cc ContainerRecycleRequested PrimaryVolumeChange { "rootPath":"//10.0.176.11/volume-22-default/d5d73fcc785e2f55ea4b/360c5523f6e9423fa57c7210e6cf73a1", "recycleType":"NonOverlapped" }

docwebsite ContainerRecycleRequested AppFrameworkVersionChange { "appFrameworkVersion":"docweb.azurecr.io/jekyll:69", "recycleType":"OverlappedWithDifferentConfig" }

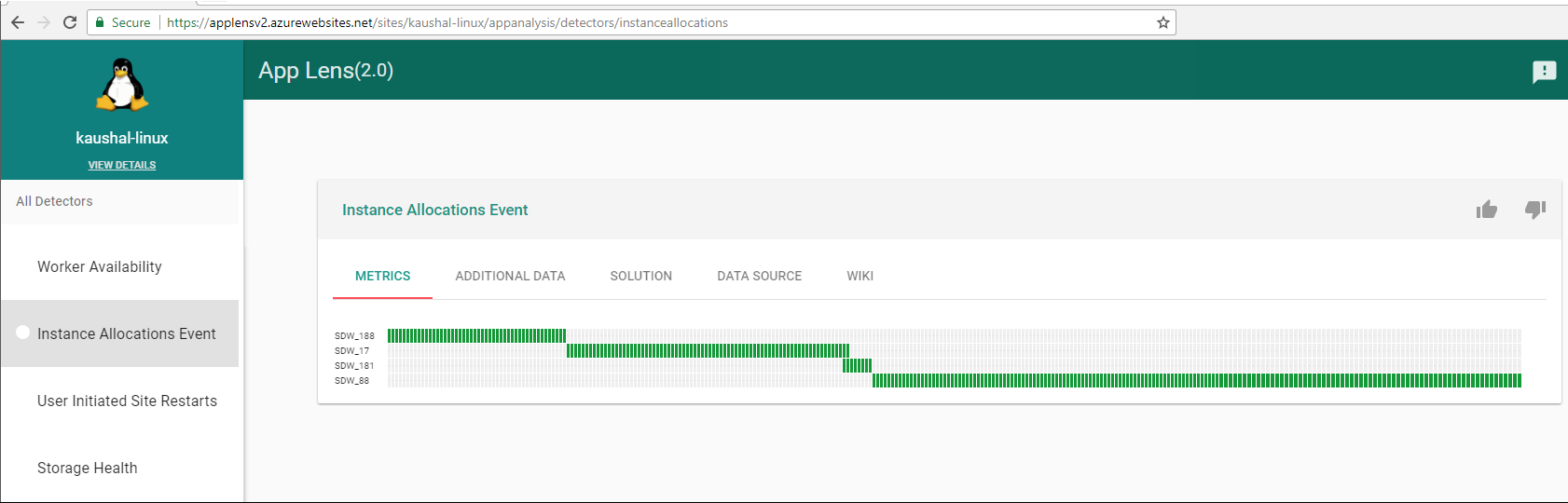
~1docwebsite ContainerRecycleRequested AppSettingsChange { "isMainSite":"False", "recycleType":"OverlappedWithDifferentConfig" }

App setting changes are always overlapped recycles and should not cause any requests to fail.

## Identifying Changed Instances

App Lens now indicates instance allocation events for apps using Linux containers.

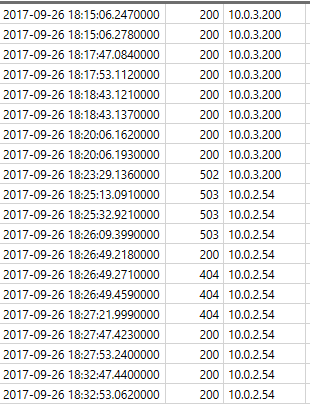
<https://applensv2.azurewebsites.net/sites/kaushal-linux/appanalysis/detectors/instanceallocations>



Right now, there is no way to find out if the web app shifted instances in App Lens. Therefore, we first start with the front-end logs table and check the ServerRouted column.

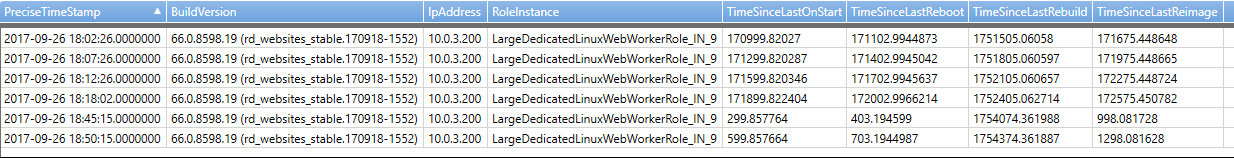
AntaresIISLogFrontEndTable  
| where PreciseTimeStamp >= datetime(2017-09-26 18:15:00) and PreciseTimeStamp <= datetime(2017-09-26 18:35:00)  
| where Cs\_host =~ "mysitename.azurewebsites.net"  
| where User\_agent != "AlwaysOn"  
| project PreciseTimeStamp, Cs\_uri\_stem, Sc\_status , ServerRouted   
| order by PreciseTimeStamp asc

You will see the ServerRouted column change if there was a switch. In the following output, this occurred at around 18.25.



Now we need to know if this was because of an infrastructure upgrade. For this we check LinuxRoleInstanceHeartBeats and we do not see any change in buildversion, but we see the instance rebooted.

LinuxRoleInstanceHeartBeats  
| where PreciseTimeStamp >= datetime(2017-09-26 18:00:00) and PreciseTimeStamp <= datetime(2017-09-26 18:55:00)  
| where IpAddress == "10.0.3.200"  
| where SourceMoniker == "LWAWSPRODAM2125"   
| project PreciseTimeStamp, BuildVersion , IpAddress , RoleInstance, TimeSinceLastOnStart , TimeSinceLastReboot, TimeSinceLastRebuild , TimeSinceLastReimage

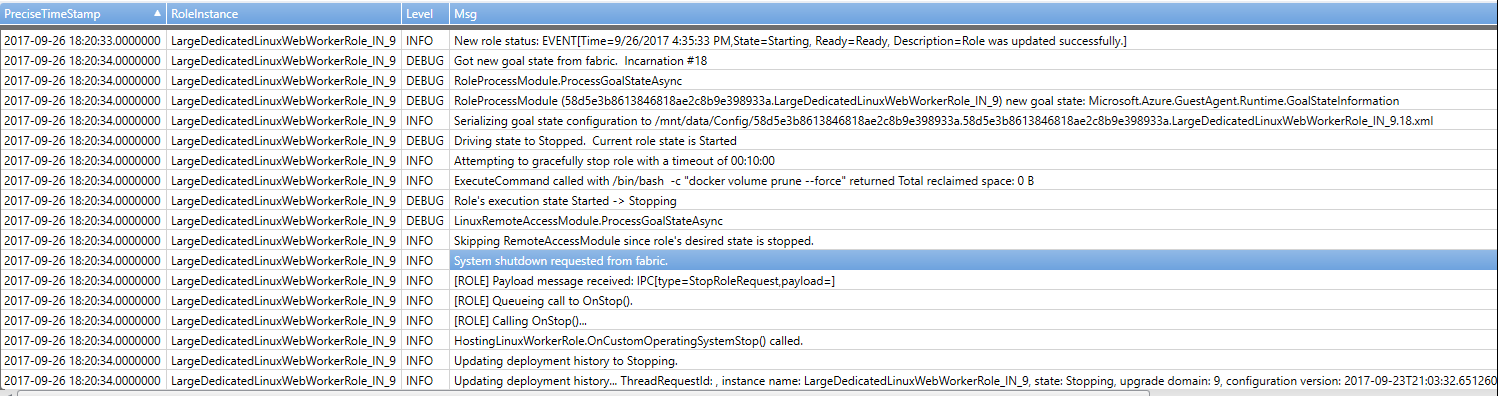


As you can see, TimeSinceLastReboot is showing that the worker was rebooted at around 18.38 and there is no BuildVersion change. Therefore, this is not related to an infrastructure upgrade.

We further check the WorkerAzureAgentEventTable, which gives more information.

WorkerAzureAgentEventTable   
| where PreciseTimeStamp >= datetime(2017-09-26 18:00:00) and PreciseTimeStamp <= datetime(2017-09-26 18:55:00)  
| where SourceMoniker == "LWAWSPRODAM2125" and RoleInstance == "LargeDedicatedLinuxWebWorkerRole\_IN\_9"

| project PreciseTimeStamp , RoleInstance , Level , Msg , Tenant



Now we see that the worker was actually requested to reboot based on a request from the fabric. Typically, instance changes should be transparent to the user. However, it does cause a cold start and sometimes customers might see some 5xx messages.

## Check for High CPU

The LinuxSiteStats table logs the CPU and memory being used per container/site.

Run: [[Kusto.Explorer](https://wawseas.kusto.windows.net:443/wawsprod?query=H4sIAAAAAAAEAPPJzCutCM4sSQ0uSSwp5uWqUSjPSC1KVQjx9HUNDnH0DVCwU0hJLEktycxN1TAyMLTQNTDTNTRTMDC1MjAAIj0DCNBUSMxLQdJlg12XOZouhIUgN%2fgl5qYq2NYpKBUnZuomFmQqgaTzi1JSixSSKpEMTyxOBskUFOVnpSaXICR04KboKDgXlAakFiWn5pUkpqcq8HIBAETL5hbpAAAA)] [[Kusto.Explorer on SAW](https://wawseas.kusto.windows.net:443/wawsprod?query=H4sIAAAAAAAEAPPJzCutCM4sSQ0uSSwp5uWqUSjPSC1KVQjx9HUNDnH0DVCwU0hJLEktycxN1TAyMLTQNTDTNTRTMDC1MjAAIj0DCNBUSMxLQdJlg12XOZouhIUgN%2fgl5qYq2NYpKBUnZuomFmQqgaTzi1JSixSSKpEMTyxOBskUFOVnpSaXICR04KboKDgXlAakFiWn5pUkpqcq8HIBAETL5hbpAAAA&saw=1)] [[Kusto.WebExplorer](https://web.kusto.windows.net/clusters/wawseas.kusto.windows.net/databases/wawsprod?q=H4sIAAAAAAAEAPPJzCutCM4sSQ0uSSwp5uWqUSjPSC1KVQjx9HUNDnH0DVCwU0hJLEktycxN1TAyMLTQNTDTNTRTMDC1MjAAIj0DCNBUSMxLQdJlg12XOZouhIUgN%2fgl5qYq2NYpKBUnZuomFmQqgaTzi1JSixSSKpEMTyxOBskUFOVnpSaXICR04KboKDgXlAakFiWn5pUkpqcq8HIBAETL5hbpAAAA)] [[Lens](https://lens.msftcloudes.com/v2/#/discover/query//results?datasource=(cluster:wawseas.kusto.windows.net,database:wawsprod,type:Kusto)&query=H4sIAAAAAAAEAPPJzCutCM4sSQ0uSSwp5uWqUSjPSC1KVQjx9HUNDnH0DVCwU0hJLEktycxN1TAyMLTQNTDTNTRTMDC1MjAAIj0DCNBUSMxLQdJlg12XOZouhIUgN%2fgl5qYq2NYpKBUnZuomFm)] <https://wawseas.kusto.windows.net:443/wawsprod>

LinuxSiteStats

| where TIMESTAMP > datetime(2018-06-16 05:00:00.0000000) and TIMESTAMP < datetime(2018-06-16 07:00:00.0000000)

| where SiteName =~ "sai-api"

| order by TIMESTAMP asc

| project TIMESTAMP, SiteName, CpuPercentage

TIMESTAMP SiteName CpuPercentage

2018-06-16 05:01:00.0000000 sai-api 73.0838167493815

2018-06-16 05:02:00.0000000 sai-api 27.1845453801574

2018-06-16 05:02:00.0000000 sai-api 23.6179998527517

2018-06-16 05:02:00.0000000 sai-api 74.5106053199789

2018-06-16 05:02:00.0000000 sai-api 13.3489398894558

2018-06-16 05:02:00.0000000 sai-api 13.8846616889632

2018-06-16 05:02:00.0000000 sai-api 18.6252878753544

2018-06-16 05:02:00.0000000 sai-api 47.522757189395

2018-06-16 05:03:00.0000000 sai-api 19.4454123759773

2018-06-16 05:03:00.0000000 sai-api 81.560980351758

As you can see, the CPU seems to be spiking for this container. If you want to check the CPU for the instance, you will need to check the WorkerSystemStats table.

WorkerSystemStats

| where TIMESTAMP > datetime(2018-06-15 20:51:00.0000000) and TIMESTAMP < datetime(2018-06-15 21:51:00.0000000)

| where RoleInstance == "SmallDedicatedLinuxWebWorkerRole\_IN\_84" and Tenant == "4885551c872a428eb63e5c654c06ed0c"

| where Facility == "top"

| order by TIMESTAMP asc

PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND

130512 231105 20 0 547940 30368 21428 D 70.6 1.8 0:16.52 apache2

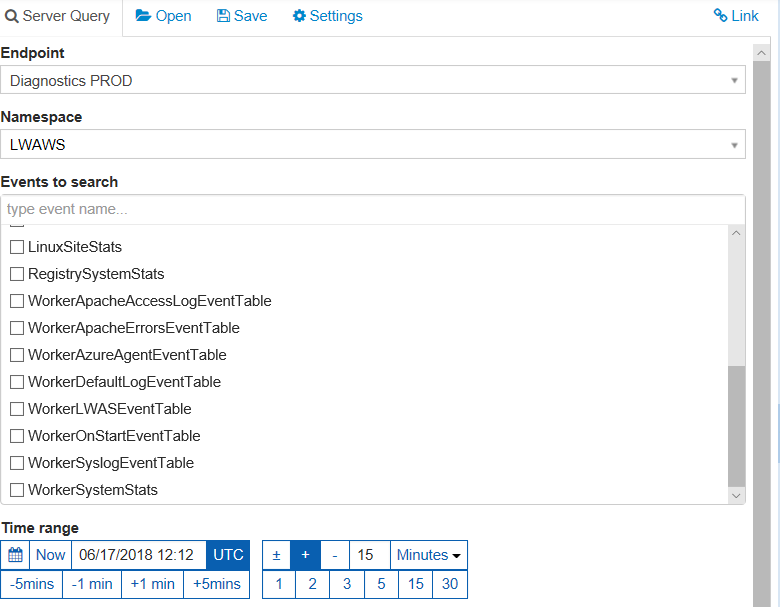
1859 231105 20 0 548280 31708 21232 R 11.8 1.9 0:03.22 apache2

2801 root 20 0 33220 2912 2612 R 5.9 0.2 0:00.01 top

It is the same logic for checking Memory usage.

## Using Jarvis

You can use Jarvis in the same way as you would for Windows Web Apps except that the namespace is LWAWS.



You will typically use Jarvis when the issue the customer reports is more than 30 days old and the information is no longer available in Kusto.

# Exercise: Experiment with Kusto

Use what you’ve learned to run some Kusto queries on the failing Web App you created in the lab earlier in this module.

**Important:** Once you are finished experimenting, delete the resource group you created in the lab.