Outbound IP Registration to Azure SQL using Azure Functions

2017-11-17 / JUSTIN YOO

As Azure SQL Database is PaaS, it has its own firewall settings. Due to its white-listed nature, only traffic from registered IP addresses is allowed to access to the server instance. Of course, there is an option that the server allows all Azure resources to access to the server. However, this is not secure because there are chances that some malicious attacks come from other Azure resources. Therefore, registering only outbound IP addresses assigned to other Azure resources like Azure Web App instances is strongly recommended. Interestingly, according to this article, those outbound IP addresses, assigned to a specific Azure Web App instance, can change from time to time when the app instance is restarted or scaling happens. If those outbound IP addresses are updated, there is no way to let the Azure SQL Database server instance know unless manually updating them. I expected that Azure Event Grid would support this scenario, but at the time of this writing, apparently it's not yet possible. However, there is still a workaround, if we use Azure Functions. In this post, I'm going to show how to update the firewall rules on an Azure SQL Database instance, using Azure Functions and Azure Fluent SDK.

Why Fluent SDK?

There is the Azure SDK library and Fluent SDK used to be a part of it. From the functionality point of view both are the same as each other. However, Fluent SDK offers more succinct way of handling Azure resources, and provides better code readability. For example, this is how to get authenticated and authorised to access to Azure resources:

```
var credentials = SdkContext.AzureCredentialsFactory
                            .FromServicePrincipal(clientId,
                                                  clientSecret,
                                                  tenantId,
                                                  AzureEnvironment.AzureGlobalCloud);
var azure = Azure.Configure()
                 .WithLogLevel(logLevel)
                 .Authenticate(credentials)
                 .WithSubscription(subscriptionId);
```

Can you see how the code looks like? It looks dead simple, yeah? With this Fluent SDK, let's move on.

Building an HTTP Trigger

If I can draw a user story reflecting this scenario, it would be:

- AS a DevOps engineer,
- GIVEN the name of Azure Resource Group,
- I WANT to retrieve all outbound IP addresses from Azure Web App instances and all firewall rules registered to Azure SQL Databases, from the resource group,
- SO THAT new IP addresses are registered to the firewall rules, as well as unused ones are deleted from the firewall rule.

First of all, for local debugging purpose, it's always a good idea to start from an HTTP trigger. Let's create a simple HTTP trigger function like:

```
public static class UpdateFirewallRulesHttpTrigger
{
    [FunctionName("UpdateFirewallRulesHttpTrigger")]
    public static async Task<HttpResponseMessage> Run(
        [HttpTrigger(AuthorizationLevel.Function, "post", Route = "firewall/rules")]
HttpRequestMessage req,
        TraceWriter log)
    {
        var resourceGroupName = Config.ResourceGroupName;

        log.Info($"Firewall rules on database servers in {resourceGroupName} are
updating...");

        log.Info($"Firewall rules on database servers in {resourceGroupName} have
been updated.");

        return req.CreateResponse(HttpStatusCode.OK);
    }
}
```

This is just a scaffolded function so it does nothing with Azure resources but leave logs onto the console.

Let's put the basic authentication logic using Fluent SDK.

```
public static class UpdateFirewallRulesHttpTrigger
{
    [FunctionName("UpdateFirewallRulesHttpTrigger")]
    public static async Task<HttpResponseMessage> Run(
        [HttpTrigger(AuthorizationLevel.Function, "post", Route =
"firewall/rules")]HttpRequestMessage req,
        TraceWriter log)
    {
        var tenantId = Config.TenantId;
        var subscriptionId = Config.SubscriptionId;
        var clientId = Config.ClientId;
        var clientSecret = Config.ClientSecret;

        var credentials =
            SdkContext.AzureCredentialsFactory
                      .FromServicePrincipal(clientId,
                                            clientSecret,
                                            tenantId,
                                            AzureEnvironment.AzureGlobalCloud);

        var logLevel = HttpLoggingDelegatingHandler.Level.Basic;
        var azure = Azure.Configure()
                         .WithLogLevel(logLevel)
                         .Authenticate(credentials)
                         .WithSubscription(subscriptionId);

        var resourceGroupName = Config.ResourceGroupName;

        log.Info($"Firewall rules on database servers in {resourceGroupName} are
updating...");

        log.Info($"Firewall rules on database servers in {resourceGroupName} have
been updated.");

        return req.CreateResponse(HttpStatusCode.OK);
    }
}
```

There are a few spots noticeable.

1. Azure credentials are handled by `SdkContext` and Fluent API.
2. Azure context is handled by `Azure` and Fluent API.
3. All environment variables are converted to `Config`, a strongly-typed object.

The `Config` is a static instance that retrieves environment variables. You can still use `ConfigurationManager.AppSettings["KEY"]` for it, but the `ConfigurationManage` won't be a good idea when Azure Functions move forward to .NET Standard.

So, it's much safer to use `Environment.GetEnvironmentVariable("KEY")`. Of course, this `Config` class and its properties might not need the `static` modifier, if you consider dependency injection. For the convenience sake, I'm sticking on the `static` nature, for now. Here's the code:

```
public static class Config
{
    public static string TenantId { get; } =
        Environment.GetEnvironmentVariable("Sp.DirectoryId");

    public static string SubscriptionId { get; } =
        Environment.GetEnvironmentVariable("Sp.SubscriptionId");

    public static string ClientId { get; } =
        Environment.GetEnvironmentVariable("Sp.ApplicationId");

    public static string ClientSecret { get; } =
        Environment.GetEnvironmentVariable("Sp.ApplicationKey");

    public static string ResourceGroupName { get; } =
        Environment.GetEnvironmentVariable("Rg.Name");
}
```

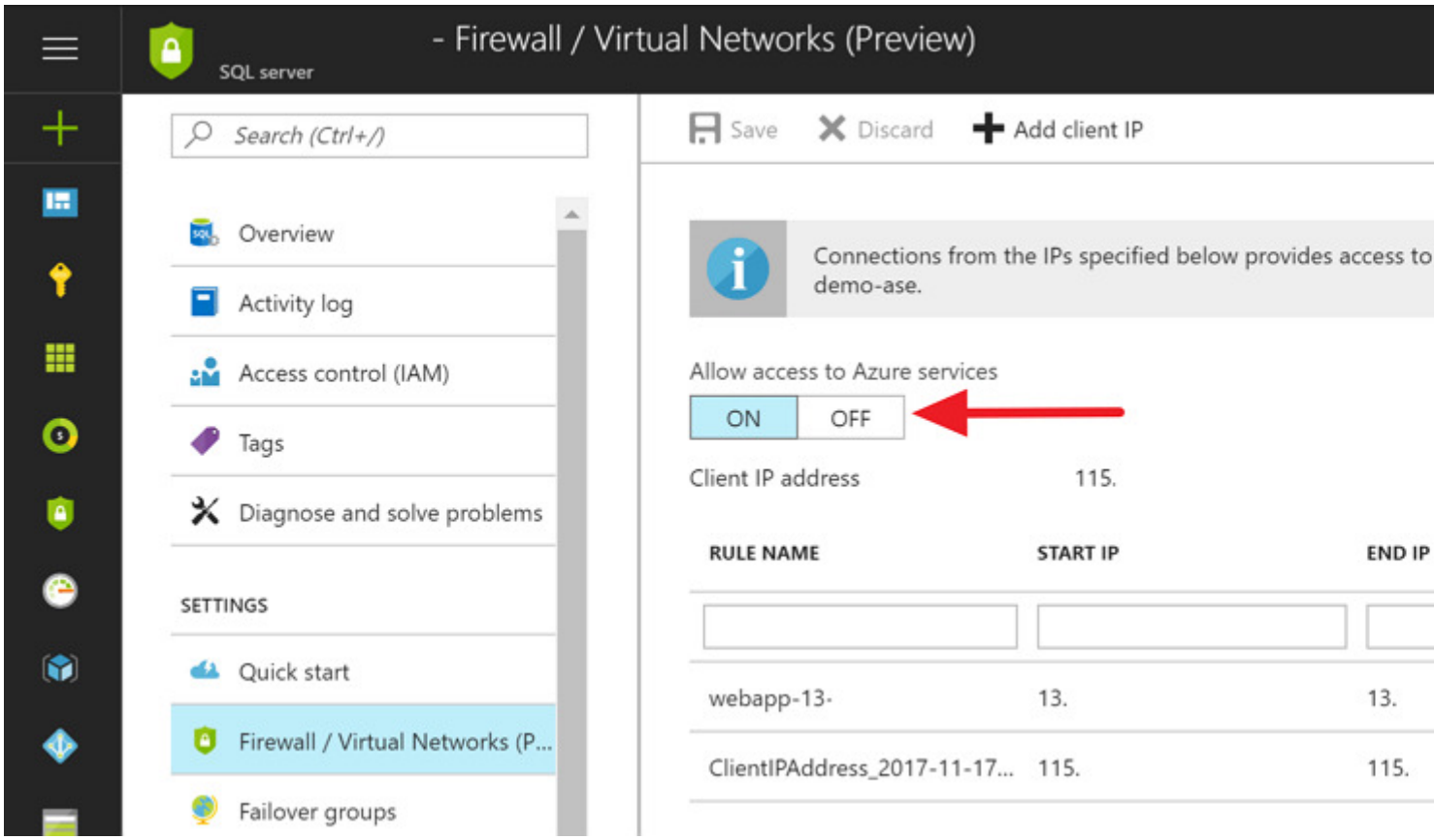
Now, we need to get outbound IP addresses from web apps in a given resource group. Between the two log lines put several lines of code to retrieve all web app instances then all outbound IP addresses are fetched from there.

```
[FunctionName("UpdateFirewallRulesHttpTrigger")]
public static async Task<HttpResponseMessage> Run(
    [HttpTrigger(AuthorizationLevel.Function, "post", Route =
"firewall/rules")]HttpRequestMessage req,
    TraceWriter log)
{
    ...

    log.Info($"Firewall rules on database servers in {resourceGroupName} are
updating...");

    var res = await azure.WebApps
                         .Inner
                         .ListByResourceGroupWithHttpMessagesAsync(resourceGroupName)
                         .ConfigureAwait(false);

    var webapps = res.Body.ToList();

    var outboundIps = webapps.SelectMany(p => p.OutboundIpAddresses.Split(','))
                             .Distinct()
                             .ToList();

    log.Info($"Firewall rules on database servers in {resourceGroupName} have been
updated.");

    return req.CreateResponse(HttpStatusCode.OK);
}
```

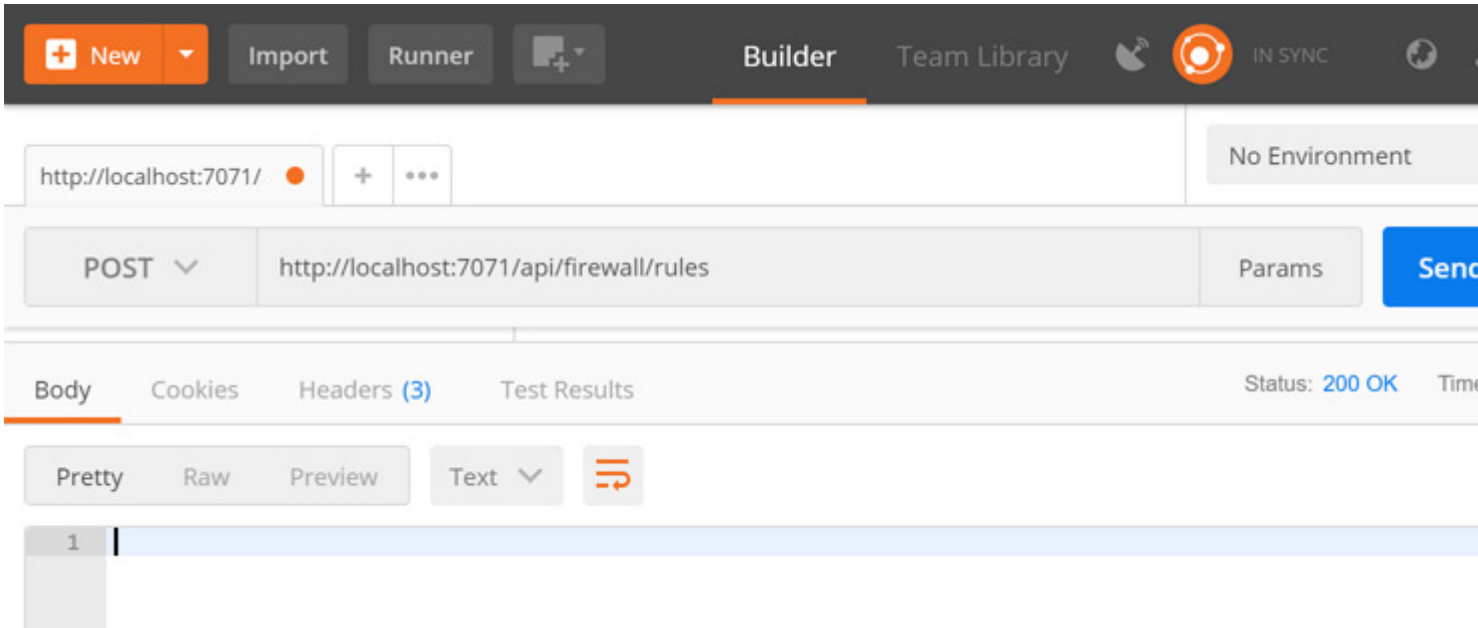
Those IP addresses need to be registered to firewall settings on each Azure SQL Database instance. If there are discrepancies between outbound IP addresses and registered IP addresses, all unnecessary IPs should be removed from the firewall rules and only newly updated IP addresses should be added to the rule. Let's finish up the function code. Once all Azure SQL Database instances are populated, code loops through them. In the loop, all registered IP addresses are fetched and compared to the outbound IPs so that we know which IP addresses are to be removed and inserted.

```
[FunctionName("UpdateFirewallRulesHttpTrigger")]
public static async Task<HttpResponseMessage> Run(
    [HttpTrigger(AuthorizationLevel.Function, "post", Route =
"firewall/rules")]HttpRequestMessage req,
    TraceWriter log)
{
    ...

    log.Info($"Firewall rules on database servers in {resourceGroupName} are
updating...");

    ...

    var tasks = new List<Task>();

    var servers = await azure.SqlServers
                             .ListByResourceGroupAsync(resourceGroupName)
                             .ConfigureAwait(false);
    foreach (var server in servers)
    {
        var registeredIps = server.FirewallRules
                                  .List()
                                  .ToDictionary(p => p.Name, p => p.StartIPAddress);
        var ipsToExclude = registeredIps.Where(p => !outboundIps.Contains(p.Value))
                                        .Select(p => p.Key)
                                        .ToList();
        var IpsToInclude = outboundIps.Where(p => !registeredIps.ContainsValue(p))
                                      .ToList();

        var tasksToExclude = ipsToExclude.Select(ip => server.FirewallRules
                                                             .DeleteAsync(ip));
        var tasksToInclude = IpsToInclude.Select(ip => server.FirewallRules
                                                             .Define($"webapp-
{ip.Replace(".", "-")}")
                                                             .WithIPAddressRange(ip,
ip)
                                                             .CreateAsync());
        tasks.AddRange(tasksToExclude);
        tasks.AddRange(tasksToInclude);
    }

    await Task.WhenAll(tasks).ConfigureAwait(false);

    log.Info($"Firewall rules on database servers in {resourceGroupName} have been
updated.");

    return req.CreateResponse(HttpStatusCode.OK);
}
```

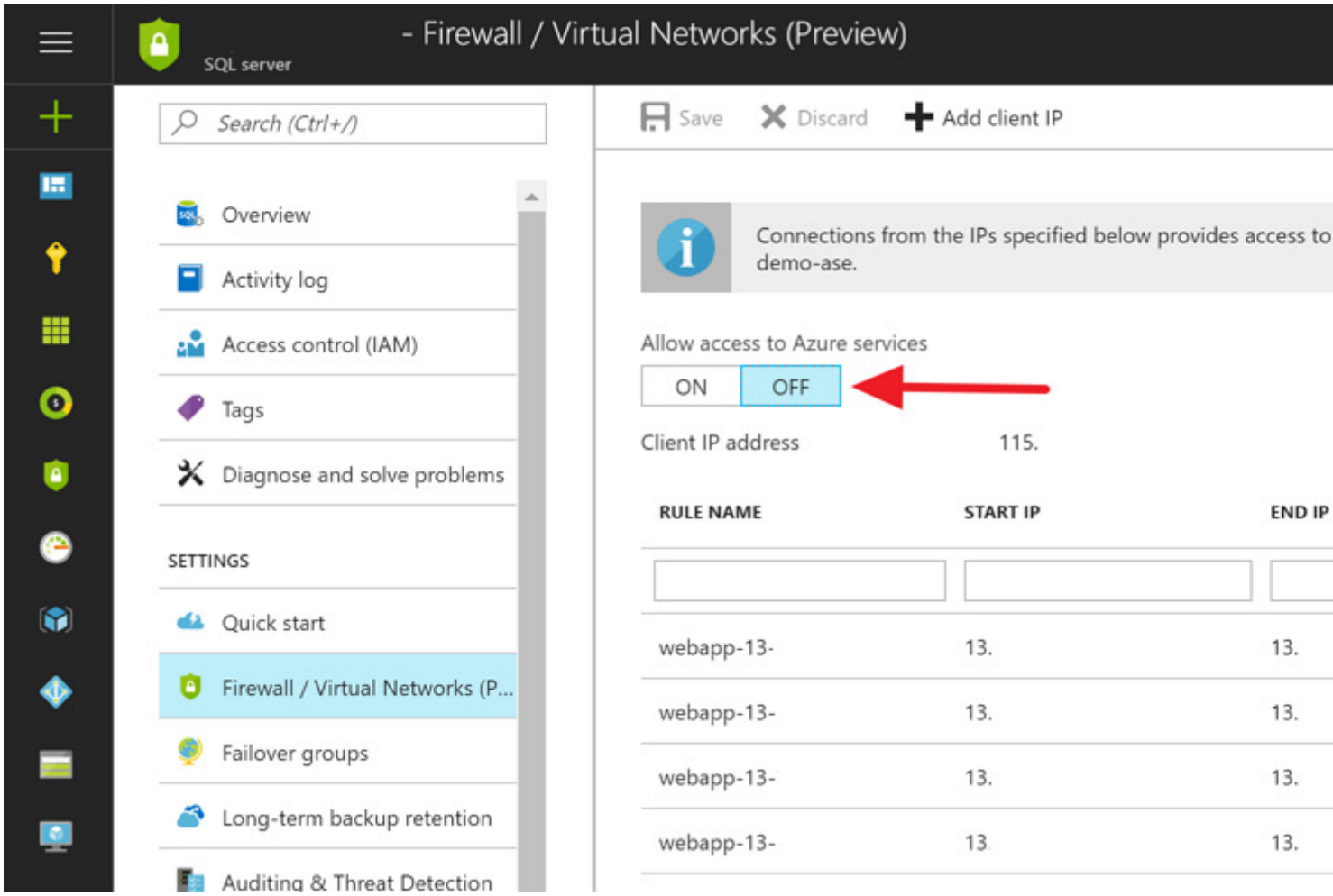
Yeah, the coding part is done. Now, let's run this on our local machine. The database instance has the following firewall rules – to allow all internal IPs from Azure resources (pointed by red arrow), one web app outbound IP (13.x.x.x) and one public IP (115.x.x.x) from my laptop.



Punch the F5 key and send an HTTP request through Postman. The function has run smoothly. Now we're expecting all the Azure internal IP addresses will be removed and my public IP will be removed, but the existing web app IP will remain.



Go back to the Azure Portal and check the firewall settings. As we expected, all internal Azure IP addresses have been blocked (pointed by red arrow), my public IP has been removed, and other outbound IPs have been registered.



Unfortunately, there is no SDK ready for Azure Database for MySQL at the time of this writing. Instead, in order to apply this approach for it, we should use REST API to register outbound IP addresses.

Converting to a Timer Trigger

Once you confirm this works fine, you can simply copy and paste all the code bits into a timer trigger function so that this is triggered in a scheduled manner. The following code snippet says the timer function is triggered once a day at midnight in UTC.

```
public static class UpdateFirewallRulesTimerTrigger
{
    [FunctionName("UpdateFirewallRulesTimerTrigger")]
    public static async Task Run(
        [TimerTrigger("0 0 0 * * *")]TimerInfo myTimer,
        TraceWriter log)
    {
        // Paste the same code here.
    }
}
```

view rawUpdateFirewallRulesTimerTrigger.cs hosted with by GitHub

---

So far, we have walked through how to check Azure Web App instances' outbound IP addresses regularly and register them into the firewall rules of Azure SQL Database instance. As I stated above, once Azure Event Grid is applied to Azure Web App, this would be much easier.