Azure Partner Quickstarts: Contributor's Guide

Table of Contents

1. What are Azure Quickstart templates

Azure Partner Quickstarts are Azure Resource Manager templates created by trusted Microsoft partners and designed to help you get started with integrated, multi-artifacts solutions rather than single applications or services on Azure.
These templates are constructed to launch solutions comprising multiple applications and services from both Microsoft and Microsoft software partners, open source and proprietary. These Quickstarts help automate much of the manual work that would be otherwise normally be involved in stitching artifacts together.

Quickstarts are available on Partner QS portal - https://partnerquickstarts.azurewebsites.net/ . This portal is also used for end to end management of partner Quickstarts lifecycle.

2. Who can contribute

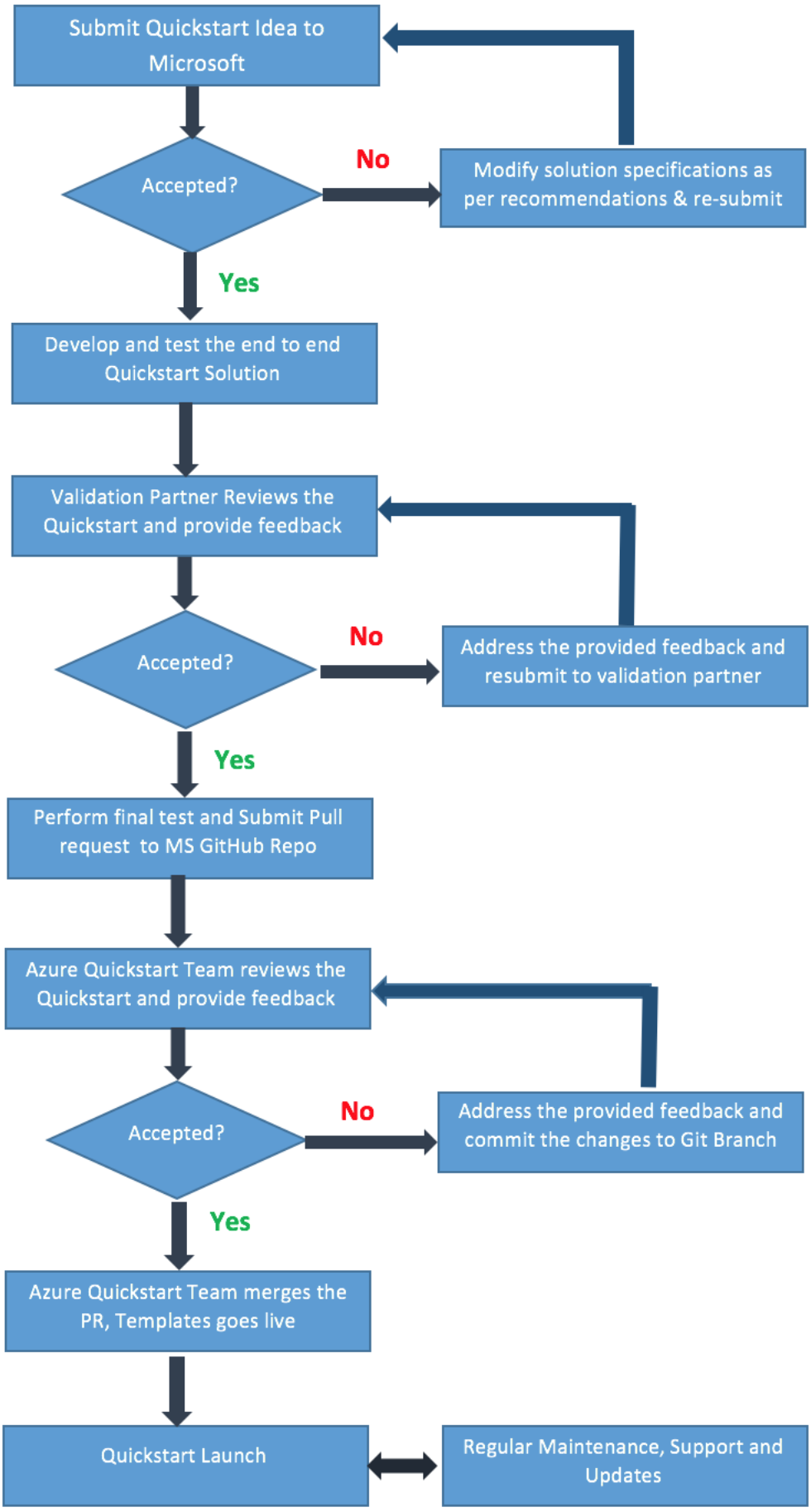
- Microsoft Partners

3. How to Contribute: Process Flow

A. Process Flowchart

Quickstart solutions goes through multiple checks and validation before getting published on azure Quickstarts from Microsoft partners portal

Following flow chart explains the high-level process to be completed for a new quickstart templates to get published. Microsoft Azure partner QuickStarts team along with quickstarts program support partner (Spektra Systems) will work with you to get your quickstart published through this process.


Submit Quickstart Idea to Microsoft
Accepted?
No
Modify solution specifications as per recommendations & re-submit
Yes
Develop and test the end to end Quickstart Solution
Validation Partner Reviews the Quickstart and provide feedback
Accepted?
No
Address the provided feedback and resubmit to validation partner
Yes
Perform final test and Submit Pull request to MS GitHub Repo
Azure Quickstart Team reviews the Quickstart and provide feedback
Accepted?
No
Address the provided feedback and commit the changes to Git Branch
Yes
Azure Quickstart Team merges the PR, Templates goes live
Quickstart Launch
Regular Maintenance, Support and Updates

B. Submit your quickstart Idea

Partners can submit their quickstart idea's by filling out the form at https://bit.ly/submitapq

An ideal submission should include

- Title of the Quickstart with brief overview of solution
- Value proposition with target geo segment
- Azure Solution Components of Quickstart
- 3rd Party/Market Place items included in the Quickstart
- Basic Solution Architecture Diagram

Microsoft would be evaluating the idea and provide feedback accordingly. Partner may or may not have to modify the QS plan based on the feedback. Once Quickstart idea is approved, Kick-off meeting will be scheduled to discuss the solution aspects and timelines for go-live of Quickstart. Quickstart program partner, Spektra Systems would be helping the quickstart development partner throughout this process

C. Quickstart Plan, Develop, Test & Document

After acceptance of the Quickstart idea, Partner may start to develop the Quickstart solution. Partner should follow below instructions.

- Understand and follow Solution Design Consideration best practices available in this document
- Understand and follow template development checklist available in this document
- Build the Quickstart template comprising
  - azuredeploy.json Template File
  - Nested templates if any
  - azuredeploy.parameters.json file
  - Documentations as specified in Section-5
- Develop the required documentation for the Quickstart following best practices given here
- Test the Quickstart in every scenario available here

Spektra systems would be working with you for:

- Validating the final QS architecture
- Template, documentation development support and validation
- Testing
- Final review and approval
- Launch Activities

D. Publish the Quickstart

Once template development partner has completed all steps in Section-C, they can move to validation phase of the solution and launch the Quickstart for public access.

- Submit the Quickstart to Azure Quickstart validation partner for validation of the solution. Follow this to learn more
- Incorporate any feedback provided by azure Quickstart validation partner and submit Pull Request to submit the Quickstart to Microsoft Azure GitHub repo.
- Following the publishing guidelines available in the document
- Add the quickstart to partner Quickstarts portal.
- Quickstart template will get launched on azure partner quickstarts portal Launch activities may also include formal announcement on social media and blogs, as applicable.

E. Support and Maintenance & Breakfix

Template development partner should be providing the post publishing support, maintenance and updates on the quickstart as and when required.

- Follow Support and Maintenance Guidelines available in this document

F. Azure Partner quickstarts Portal

Template development partner should be using partner quickstart portal (https://partnerquickstarts.azurewebsites.net) for end to end lifecycle management of the Quickstarts.

- Request login for the portal if not available already by writing to quickstartssupport@spektrasystems.com

Portal can be used for:

- View, Manage and modify existing contributed quickstart
- Submit new quickstart
- Verify last validation reports.
- Update Status if QS validation fails.

4. Quickstart Technical Solution Best Practices

A. Envision the Quickstart architecture

First step towards building a good Quickstart would be to draw the overall low-level architecture of the solution. You should start developing template once this architecture

design is finalized. Architecture design should contain overall solution components such as Virtual Network, Subnet's, Storage accounts, Virtual Machines, App Service etc. Sample architecture diagram can be found here.

B. Azure Region Support

Your Quick Start should be deployable across most Azure Regions. You should verify availability of services in azure regions for your template and update documentation accordingly. Services availability by region is listed here.

C. Identify the Outside and Inside of a VM

As you design your template, it's helpful to look at the requirements in terms of what's outside and inside of the virtual machines (VMs):

- Outside means the VMs and other resources of your deployment, such as the network topology, tagging, references to the certs/secrets, and role-based access control. All of them are part of your ARM template.
- For the VM's insides—that is, the installed software and overall desired state configuration—other mechanisms are used in whole or in part, such as VM extensions or scripts. These may be identified and executed by the template but aren't in it.

Common examples of activities you would do "inside the box" include

- Install or remove server roles and features
- Install and configure software at the node or cluster level
- Deploy websites on a web server
- Deploy database schemas
- Manage registry or other types of configuration settings
- Manage files and directories
- Start, stop, and manage processes and services
- Manage local groups and user accounts
- Install and manage packages (.msi, .exe, yum, etc.)
- Manage environment variables
- Run native scripts (Windows PowerShell, bash, etc.)

Typically, you will use one or more VM extension to achieve automatic provisioning of inside VM components. Most widely used VM Extension will include

- Desired State Configuration (DSC)
- Custom Script Extension (For both Windows and Linux VM's)
- Other 3rd part VM Extension such as Chef/Puppet etc.

Defining inside and outside VM things will help you in designing the solution template effectivity.

D. Choosing free-form vs. known configurations

You might initially think a template should give consumers the utmost flexibility, but many considerations affect the choice of whether to use free-form configurations vs. known configurations. This section identifies the key customer requirements and technical considerations that shaped the approach shared in this document.

Free-form configurations:

On the surface, free-form configurations sound ideal. They allow you to select a VM type and provide an arbitrary number of nodes and attached disks for those nodes — and do so as parameters to a template. However, this approach is not ideal for some scenarios.
In Sizes for virtual machines, the different VM types and available sizes are identified, and each of the number of durable disks (2, 4, 8, 16, or 32) that can be attached. Each attached disk provides 500 IOPS and multiples of these disks can be pooled for a multiplier of that number of IOPS. For example, 16 disks can be pooled to provide 8,000 IOPS. Pooling is done with configuration in the operating system, using Microsoft Windows Storage Spaces or redundant array of inexpensive disks (RAID) in Linux.
A free-form configuration enables the selection several VM instances, various VM types and sizes for those instances, various disks for the VM type, and one or more scripts to configure the VM contents. It is common that a deployment may have multiple types of nodes, such as master and data nodes, so this flexibility is often provided for every node type.
As you start to deploy clusters of any significance, you begin to work with these complex scenarios. If you were deploying a Hadoop cluster, for example, with 8 master nodes and 200 data nodes, and pooled 4 attached disks on each master node and pooled 16 attached disks per data node, you would have 208 VMs and 3,232 disks to manage.

A storage account will throttle requests above its identified 20,000 transactions/second limit, so you should look at storage account partitioning and use calculations to determine the appropriate number of storage accounts to accommodate this topology. Given the multitude of combinations supported by the free-form approach, dynamic calculations are required to determine the appropriate partitioning. The Azure Resource Manager Template Language does not presently provide mathematical functions, so you must perform these calculations in code, generating a unique, hard-coded template with the appropriate details.

In enterprise IT and SI scenarios, someone must maintain the templates and support the deployed topologies for one or more organizations. This additional overhead — different configurations and templates for each customer — is far from desirable.

Considering all these factors, a truly free-form configuration is less appealing than at first blush.

Known configurations - the t-shirt sizing approach:

Rather than offer a template that provides total flexibility and countless variations, in our experience a common pattern is to provide the ability to select known configurations — in effect, standard t-shirt sizes such as sandbox, small, medium, and large. Other examples of t-shirt sizes are product offerings, such as community edition or enterprise edition. In other cases, it may be workload-specific configurations of a technology – such as map reduce or no sql.
Many enterprise IT organizations, OSS vendors, and SIs make their offerings available today in this way in on-premises, virtualized environments (enterprises) or as software-as-a-service (SaaS) offerings (CSVs and OSVs).
This approach provides good, known configurations of varying sizes that are preconfigured for customers. Without known configurations, end customers must determine cluster sizing on their own, factor in platform resource constraints, and do math to identify the resulting partitioning of storage accounts and other resources (due to cluster size and resource constraints). Known configurations enable customers to easily select the right t-shirt size—that is, a given deployment. In addition to making a better experience for the customer, a small number of known configurations is easier to support and can help you deliver a higher level of density.

A known configuration approach focused on t-shirt sizes may also have varying number of nodes within a size. For example, a small t-shirt size may be between 3 and 10 nodes. The t-shirt size would be designed to accommodate up to 10 nodes and provide the consumer the ability to make free form selections up to the maximum size identified.
A t-shirt size based on workload type, may be more free form in nature in terms of the number of nodes that can be deployed but will have workload distinct node size and configuration of the software on the node.
T-shirt sizes based on product offerings, such as community or Enterprise, may have distinct resource types and maximum number of nodes that can be deployed, typically tied to licensing considerations or feature availability across the different offerings.
You can also accommodate customers with unique variants using the JSON-based templates. When dealing with outliers, you can incorporate the appropriate planning and considerations for development, support, and costing. Based on the customer template consumption scenarios, and requirements identified at the start of this document, we identified a pattern for template decomposition.

E. Designing Modularity with Nested Templates

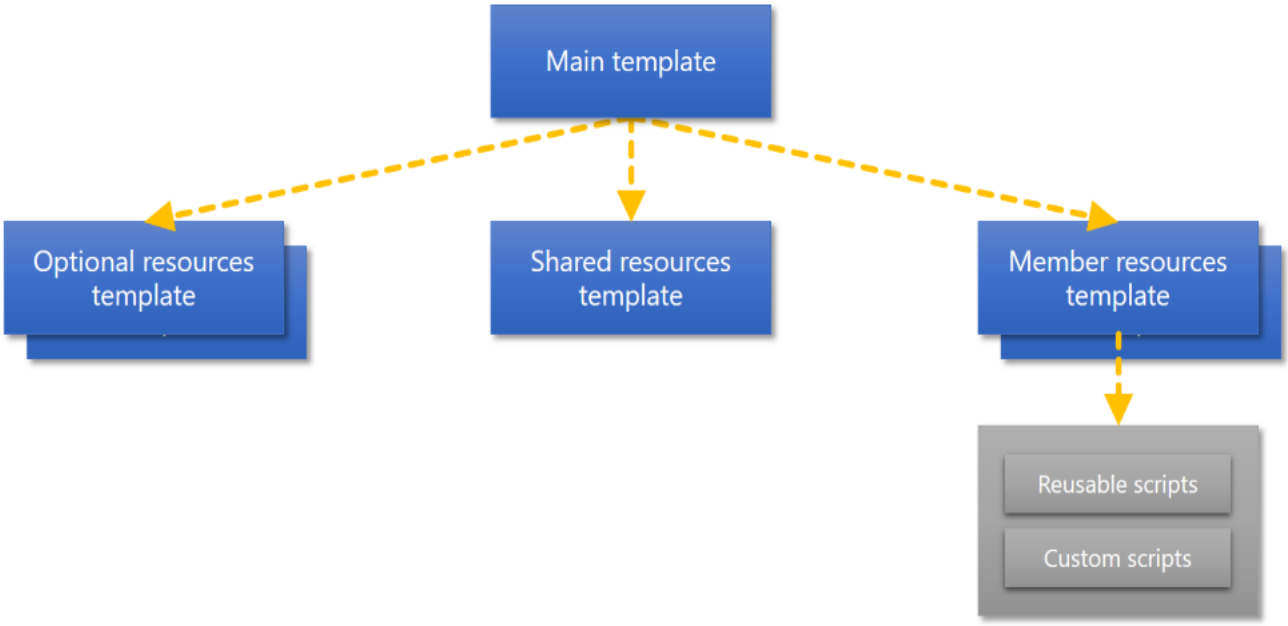
To deploy your solution, you can use either a single template or a main template with multiple nested templates. Nested templates are common for more advanced scenarios. Nested templates contain the following advantages:

- Provides benefits in terms of testing, reuse, and readability.
- Can reuse nested templates with other main templates of different quickstart solutions.

When you decide to decompose your template design into multiple nested templates, the following guidelines helps in standardize the design. The recommended design consists of the following templates:

F. **Main** template (azuredeploy.json): Used for the input parameters.

- **Shared resources template**: Deploys the shared resources that all other resources use (for example, virtual network, availability sets). The expression **dependsOn** enforces that this template is deployed before the other templates.
- **Optional resources template:** Conditionally deploys resources based on a parameter (for example, a jumpbox)
- **Member resources templates**: Each instance type within an application tier has its own configuration. Within a tier, different instance types can be defined (such as, first instance creates a cluster, additional instances are added to the existing cluster). Each instance type has its own deployment template.
- **Scripts**: Widely reusable scripts are applicable for each instance type (for example, initialize and format additional disks). Custom scripts are created for specific customization purpose are different per instance type.



You can follow below articles to learn more about passing values and sharing state between templates:

- https://docs.microsoft.com/en-us/azure/azure-resource-manager/best-practices-resource-manager-state
- https://docs.microsoft.com/en-us/azure/azure-resource-manager/resource-group-linked-templates

G. Identify and build template Pre-requisite (if any)

If you're building a template which needs some existing resources in subscription to complete the deployment, it is recommended to create a pre-requisite azure deployment template. Some of the examples of existing resources includes

- Virtual Network
- Storage Account
- Key vault

Some users may or may not have these existing resources ready, which may lead to a deployment failure of the solution template. To avoid this, it is recommended to create a azureprereqdeploy.json ARM template which can be deployed before the actual Quickstart to avoid deployment failure of main Quickstart solution due to existing resources. Partner should also add this is documentation.
Considerations:

- Identify Number and type of pre-req components for the Quickstart
- Create ARM template to create the existing resources
- Output the name's etc. values in the pre-req template deployment
- Update the documentation/process to share these details to the users who doesn't have existing resources created already.

H. Working with Unique Resource Name's

There are generally three types of resource names you work with:

- Resource names that must be unique across the azure cloud.
- Resource names that do not need to be unique but you want to provide a name that helps identify the context.
- Resource names that can be generic.

Unique resource names

You must provide a unique resource name for any resource type that has a data access endpoint. Some common types that require a unique name include:

- Storage account
- Web site
- SQL server
- Key vault
- Redis cache
- Etc.

Furthermore, storage account names must be lower-case, 24 characters or less, and not include any hyphens.

If you provide a parameter for these resource names, you must guess a unique name during deployment. Instead, you can create a variable that uses the uniqueString() function to generate a name. Frequently, you also want to add a prefix or postfix to the uniqueString result

so you can more easily determine the resource type by looking at the name. For example, you generate a unique name for a storage account with the following variable:

```
"variables": {
"storageAccountName": "[concat(uniqueString(resourceGroup().id),'storage')]"
}
```

You may want to look at azure naming convention best practices available on below links

- https://docs.microsoft.com/en-us/azure/virtual-machines/virtual-machines-windows-infrastructure-naming-guidelines?toc=%2fazure%2fvirtual-machines%2fwindows%2ftoc.json
- https://docs.microsoft.com/en-us/azure/guidance/guidance-naming-conventions

I. Identifying and defining dependencies between resources

For a given resource, there can be other resources that must exist before the resource is deployed. For example, a SQL server must exist before attempting to deploy a SQL database. You define this relationship by marking one resource as dependent on the other resource. You define a dependency with the dependsOn element, or by using the reference function.

Resource Manager evaluates the dependencies between resources, and deploys them in their dependent order. When resources are not dependent on each other, Resource Manager deploys them in parallel. You only need to define dependencies for resources that are deployed in the same template.

dependsOn

Within your template, the dependsOn element enables you to define one resource as a dependent on one or more resources. Its value can be a comma-separated list of resource names.

When defining dependencies, you can include the resource provider namespace and resource type to avoid ambiguity. For example, to clarify a load balancer and virtual network that may have the same names as other resources, use the following format:

```
"dependsOn": [
  "[concat('Microsoft.Network/loadBalancers/', variables('loadBalancerName'))]",
  "[concat('Microsoft.Network/virtualNetworks/', variables('virtualNetworkName'))]"
]
```

While you may be inclined to use dependsOn to map relationships between your resources, it's important to understand why you're doing it. For example, to document how resources are interconnected, dependsOn is not the right approach. You cannot query which resources were defined in the dependsOn element after deployment. By using dependsOn, you potentially impact deployment time because Resource Manager does not deploy in parallel two resources that have a dependency. To document relationships between resources, instead use resource linking.

Dependencies Recommendations

- Set as few dependencies as possible.
- Set a child resource as dependent on its parent resource.
- Use the **reference** function to set implicit dependencies between resources that need to share a property. Do not add an explicit dependency (**dependsOn**) when you have already defined an implicit dependency. This approach reduces the risk of having unnecessary dependencies.

- Set a dependency when a resource cannot be created without functionality from another resource. Do not set a dependency if the resources only interact after deployment.
- Let dependencies cascade without setting them explicitly. For example, your virtual machine depends on a virtual network interface, and the virtual network interface depends on a virtual network and public IP addresses. Therefore, the virtual machine is deployed after all three resources, but do not explicitly set the virtual machine as dependent on all three resources. This approach clarifies the dependency order and makes it easier to change the template later.
- If a value can be determined before deployment, try deploying the resource without a dependency. For example, if a configuration value needs the name of another resource, you might not need a dependency. This guidance does not always work because some resources verify the existence of the other resource. If you receive an error, add a dependency.

Resource Manager identifies circular dependencies during template validation. If you receive an error stating that a circular dependency exists, evaluate your template to see if any dependencies are not needed and can be removed. If removing dependencies does not work, you can avoid circular dependencies by moving some deployment operations into child resources that are deployed after the resources that have the circular dependency

J. Condition based templates deployment

You can link to different templates by passing in a parameter value that is used to construct the URI of the linked template. This approach works well when you need to specify during deployment the linked template to use. For example, you can specify one template to use for an existing storage account, and another template to use for a new storage account.

The following example shows a parameter for a storage account name, and a parameter to specify whether the storage account is new or existing.

```
"parameters": {
    "storageAccountName": {
        "type": "String"
    },
    "newOrExisting": {
        "type": "String",
        "allowedValues": [
            "new",
            "existing"
        ]
    }
},
```

You create a variable for the template URI that includes the value of the new or existing parameter

```
"variables": {
    "templatelink":
"[concat('https://raw.githubusercontent.com/exampleuser/templates/master/',parameters
('newOrExisting'),'StorageAccount.json')]"
},
```

You provide that variable value for the deployment resource

```
"resources": [
    {
        "apiVersion": "2015-01-01",
        "name": "linkedTemplate",
        "type": "Microsoft.Resources/deployments",
"properties": {
            "mode": "incremental",
            "templateLink": {
                "uri": "[variables('templatelink')]",
                "contentVersion": "1.0.0.0"
            },
            "parameters": {
                "StorageAccountName": {
                    "value": "[parameters('storageAccountName')]"
                }
            }
        }
    }
],
```

The URI resolves to a template named
either **existingStorageAccount.json** or **newStorageAccount.json**. Create templates for those URIs. The following example shows the **existingStorageAccount.json** template.

```
"{
  "$schema": "https://schema.management.azure.com/schemas/2015-01-
01/deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
  "parameters": {
    "storageAccountName": {
      "type": "String"
    }
  },
"variables": {},
  "resources": [],
  "outputs": {
    "storageAccountInfo": {
      "value": "[reference(concat('Microsoft.Storage/storageAccounts/',
parameters('storageAccountName')),providers('Microsoft.Storage',
'storageAccounts').apiVersions[0])]",
      "type" : "object"
    }
  }
}
```

The next example shows the **newStorageAccount.json** template. Notice that like the existing storage account template the storage account object is returned in the outputs. The master template works with either linked template.

```
{
  "$schema": "https://schema.management.azure.com/schemas/2015-01-
01/deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
  "parameters": {
    "storageAccountName": {
             "type": "string"
       }
  },
resources": [
    {
      "type": "Microsoft.Storage/storageAccounts",
      "name": "[parameters('StorageAccountName')]",
      "apiVersion": "2016-01-01",
      "location": "[resourceGroup().location]",
sku": {
        "name": "Standard_LRS"
},
      "kind": "Storage",
      "properties": {
      }
    }
  ],
  "outputs": {
    "storageAccountInfo": {
      "value": "[reference(concat('Microsoft.Storage/storageAccounts/',
parameters('StorageAccountName')),providers('Microsoft.Storage',
'storageAccounts').apiVersions[0])]",
      "type" : "object"
    }
  }
}
```

K. High Availability Best Practices

- Create availability set for Virtual Machines. Recommended even if you're creating single VM delivering the service, this allows users to add additional VM of same service in future easily.
- Use GRS as default type for storage account replication.
- Considering adding Load balancer to your Quickstart solution wherever applicable.

L. Use Key Vault to pass secure parameter value during deployment

When you need to pass a secure value (like a password) as a parameter during deployment, you can retrieve the value from an Azure Key Vault. You retrieve the value by referencing the key vault and secret in your parameter file. The value is never exposed because you only reference its key vault ID. You do not need to manually enter the value for the secret each time you deploy the resources. The key vault can exist in a different subscription than the resource group you are deploying to. When referencing the key vault, you include the subscription ID.

You should plan to use Key Vault to enhance the security of overall solutions. This could be useful in scenarios of complex Quickstarts passing credentials or other secrets between resources. Learn more about Key Vault in ARM Templates here.

M. Creating Multiple instances of resources

copy, copyIndex, and length

- Within the resource to create multiple times, you can define a copy object that specifies the number of times to iterate. The copy takes the following format

```
"copy": {
    "name": "websites copy",
    "count": "[parameters('count')]"
}
```

- You can access the current iteration value with the copyIndex() function. The following example uses copyIndex with the concat function to construct a name.

```
[concat('examplecopy-', copyIndex())]
```

- When creating multiple resources from an array of values, you can use the length function to specify the count. You provide the array as the parameter to the length function

```
"copy": {
    "name": "websitescopy",
    "count": "[length(parameters('siteNames'))]"
}
"resources": [
  {
    "type": "{provider-namespace-and-type}"
"name": "parentResource",
    "copy": {
      /* yes, copy can be applied here */
    },
    "properties": {
      "exampleProperty": {
        /* no, copy cannot be applied here */
      } },
"resources": [
      {
        "type": "{provider-type}",
"name": "childResource",
        /* copy can be applied if resource is promoted to top level */
      }
    ]
  } ]
```

- Although you cannot apply **copy** to a property, that property is still part of the iterations of the resource that contains the property. Therefore, you can use copyIndex() within the property to specify values.

- There are several scenarios where you might want to iterate on a property in a resource. For example, you may want to specify multiple data disks for a virtual machine. To see how to iterate on a property, see Create multiple instances when copy won't work.

Use Index value in name

You can use the copy operation create multiple instances of a resource that are uniquely named based on the incrementing index. For example, you might want to add a unique number to the end of each resource name that is deployed. To deploy three web sites named:

- examplecopy-0
- examplecopy-1
- examplecopy-2.

Use the following template

```
"parameters": {
  "count": {
    "type": "int",
    "defaultValue": 3
  }
},
"resources": [
  {
      "name": "[concat('examplecopy-', copyIndex())]",
      "type": "Microsoft.Web/sites",
      "location": "East US",
      "apiVersion": "2015-08-01",
      "copy": {
         "name": "websitescopy",
         "count": "[parameters('count')]"
      },
      "properties": {
          "serverFarmId": "hostingPlanName"
      }
  }
]
```

Offset index value

In the preceding example, the index value goes from zero to 2. To offset the index value, you can pass a value in the **copyIndex()** function, such as **copyIndex(1)**. The number of iterations to perform is still specified in the copy element, but the value of copyIndex is offset by the specified value. So, using the same template as the previous example, but specifying **copyIndex(1)** would deploy three web sites named:

- examplecopy-1

- examplecopy-2
- examplecopy-3

Use Copy with Array

The copy operation is helpful when working with arrays because you can iterate through each element in the array. To deploy three web sites named

- examplecopy-Contoso
- examplecopy-Fabrikam
- examplecopy-Coho

Use the following template:

```
"parameters": {
  "org": {
     "type": "array",
     "defaultValue": [
         "Contoso",
         "Fabrikam",
         "Coho"
      ]
  }
},
"resources": [
  {
      "name": "[concat('examplecopy-', parameters('org')[copyIndex()])]",
      "type": "Microsoft.Web/sites",
      "location": "East US",
      "apiVersion": "2015-08-01",
      "copy": {
         "name": "websitescopy",
         "count": "[length(parameters('org'))]"
      },
      "properties": {
          "serverFarmId": "hostingPlanName"
      }
  }
```

N. Using Market place items in templates

You should identify all the marketplace items used in the Quickstart and their licensing consideration. Using marketplace products have following consideration.

- You should always use marketplace image of any 3rd party product used in the Quickstart solution wherever available, instead of installing it manually using custom script extension. This simplifies the licenses and pricing of third party products
- Marketplace items requires programmatic deployment of marketplace item to be enabled in the subscription. You can do this by deploying the marketplace item in subscription manually via portal once.

- Marketplace items will require a payment method associated with subscriptions
- Ensure that Template Documentation describe all marketplace items used in the quickstart
- You must include Unique GUID as tag in all template where marketplace product is used. Reach out to quickstartssupport@spektrasystems.com if you don't not have GUID available for marketplace product being used in solutions

O. Identify Post Deployment Steps

You should identify all post deployment steps which user may be required to perform after the deployment. Purpose of post deployment steps may include any of below, but not limited to these

- Access the deployed Services
- Basic Usage(Services in Action)
- Verify deployment

P. Development Tool

ARM Quickstart template can be developed in any text editor, however it is recommended to use Visual Studio/Visual Studio Code. Visual Studio provides out of the box ARM Template development utilities.

5. Template Development Checklist

A. Template Parameters Checklist

- ✓ Minimize parameters whenever possible. If you can use a variable or a literal, do so. Only provide parameters for:
  - Settings you wish to vary by environment (such as sku, size, or capacity).
  - Resource names you wish to specify for easy identification.
  - Values you use often to complete other tasks (such as admin user name).
  - Secrets (such as passwords)
  - The number or array of values to use when creating multiple instances of a resource type.
- ✓ Parameter names should follow **camelCasing**.
- ✓ Provide a description in the metadata for every parameter
- ✓ Define default values for parameters (except for passwords and SSH keys).
- ✓ Use securestring for all passwords and secrets.
- ✓ Any resources that need to be setup outside the template should be named prefixed with existing (e.g. existingVNET, existingDiagnosticsStorageAccount).
- ✓ When possible, avoid using a parameter to specify the location. Instead, use the location property of the resource group. By using the resourceGroup().location expression for all your

resources, the resources in the template are deployed in the same location as the resource group.

- ✓ If a resource type is supported in only a limited number of locations, consider specifying a valid location directly in the template. If you must use a location parameter, share that parameter value as much as possible with resources that are likely to be in the same location. This approach minimizes users having to provide locations for every resource type.
- ✓ Avoid using a parameter or variable for the API version for a resource type. Resource properties and values can vary by version number. Intellisense in code editors is not able to determine the correct schema when the API version is set to a parameter or variable. Instead, hard-code the API version in the template.

- ✓ Do not create a parameter for a **storage account name**. Storage account names need to be lower case and can't contain hyphens (-) in addition to other domain name restrictions. A storage account has a limit of 24 characters. They also need to be globally unique. To prevent any validation issue, configure a variable (using the expression **uniqueString** and a static value **storage**). Storage accounts with a common prefix (uniqueString) will not get clustered on the same racks.

```
"variables": {
     "storage": {
         "name": "[concat(uniqueString(resourceGroup().id),'storage')]",
         "type": "Standard_LRS"
     }
 },
 "resources": [
   {
       "type": "Microsoft.Storage/storageAccounts",
       "name": "[variables('storage').name]",
       "apiVersion": "2016-01-01",
       "location": "[resourceGroup().location]",
       "sku": {
           "name": "[variables('storage').type]"
       },
       ...
   }
 ]
```

- ✓ For many resources with a resource group, a name is not often relevant and using something a hard-coded string "availabilitySet" may be acceptable. You can also use variables for the name of a resource and generate names for resources with globally unique names.
  Use **displayName** tags for a "friendly" name in the JSON outline view. This should ideally match the name property value or property name.

```
"resources": [
 {
   "name": "availabilitySet",
   "type": "Microsoft.Compute/availabilitySets",
   "apiVersion": "2015-06-15",
   "location": "[resourceGroup().location]",
   "tags": { "displayName": "appTierAS" },
   "properties": {
      ...
   }
}
]
```

B. Template Variables Checklist

- ✓ Use variables for values that you need to use more than once in a template. If a value is used only once, a hard-coded value makes your template easier to read.

- ✓ You cannot use the reference function in the variables section. The reference function derives its value from the resource's runtime state, but variables are resolved during the initial parsing of the template. Instead, construct values that need the reference function directly in the resources or outputs section of the template.

- ✓ Name variables using this scheme templateScenarioResourceName (e.g. simpleLinuxVMVNET, userRoutesNSG, elasticsearchPublicIP etc.) that describe the scenario rather. This ensures when a user browses all the resources in the Portal there aren't a bunch of resources with the same name (e.g. myVNET, myPublicIP, myNSG)

- ✓ Include variables for resource names that need to be unique, as shown in Resource names.

- ✓ You can group variables into complex objects. You can reference a value from a complex object in the format variable.subentry. Grouping variables helps you track related variables and improves readability of the template.

C. Template Resources Checklist

- ✓ Use resourceGroup().location for resource locations to be compatible with all clouds

- ✓ Specify **comments** for each resource in the template to help other contributors understand the purpose of the resource.

```
"resources": [
   {
       "name": "[variables('storageAccountName')]",
       "type": "Microsoft.Storage/storageAccounts",
       "apiVersion": "2016-01-01",
       "location": "[resourceGroup().location]",
       "comments": "This storage account is used to store the VM disks",
       ...
   }
 ]
```

- ✓ Use tags to add metadata to resources that enable you to add additional information about your resources. For example, you can add metadata to a resource for billing detail purposes. For more information, see Using tags to organize your Azure resources. Specify tags properties below comments line.
- ✓ Following tags must be specified in all azure partner quickstarts template. Provider is GUID of marketplace product being used in the solution. Write to quickstartssuppport@spektrasystems.com if you don't have the GUID available.

```
"tags": {
        "quickstartName": "QSNAME",
        "provider": "Marketplace product GUID"
      },
```

- ✓ If you use a **public endpoint** in your template (such as a blob storage public endpoint), **do not hardcode** the namespace. Use the **reference** function to retrieve the namespace dynamically. This approach allows you to deploy the template to different public namespace environments, without manually changing the endpoint in the template. Set the apiVersion to the same version you are using for the storageAccount in your template.

```
"osDisk": {
     "name": "osdisk",
     "vhd": {
         "uri": "[concat(reference(concat('Microsoft.Storage/storageAccounts/',
variables('storageAccountName')), '2016-01-01').primaryEndpoints.blob,
variables('vmStorageAccountContainerName'), '/',variables('OSDiskName'),'.vhd')]"
     }
 }
```

- ✓ If the storage account is deployed in the same template, you do not need to specify the provider namespace when referencing the resource. The simplified syntax is:

```
"osDisk": {
     "name": "osdisk",
     "vhd": {
         "uri": "[concat(reference(variables('storageAccountName'), '2016-01-
01').primaryEndpoints.blob, variables('vmStorageAccountContainerName'),
'/',variables('OSDiskName'),'.vhd')]"
     }
 }
```

- ✓ If you have other values in your template configured with a public namespace, change these values to reflect the same reference function. For example, the storageUri property of the virtual machine diagnosticsProfile. You can also reference an existing storage account in a different resource group.

```
diagnosticsProfile": {
     "bootDiagnostics": {
         "enabled": "true",
         "storageUri": "[reference(concat('Microsoft.Storage/storageAccounts/',
variables('storageAccountName')), '2016-01-01').primaryEndpoints.blob]"
     }
 }
```

```
"osDisk": {
     "name": "osdisk",
     "vhd": {
         "uri":"[concat(reference(resourceId(parameters('existingResourceGroup'),
'Microsoft.Storage/storageAccounts/', parameters('existingStorageAccountName')),
'2016-01-01').primaryEndpoints.blob,  variables('vmStorageAccountContainerName'),
'/', variables('OSDiskName'),'.vhd')]"
     }
 }
```

- ✓ Assign publicIPAddresses to a virtual machine only when required for an application. To connect for debug, management or administrative purposes, use either inboundNatRules, virtualNetworkGateways or a jumpbox.
- ✓ The **domainNameLabel** property for publicIPAddresses must be unique. domainNameLabel is required to be between 3 and 63 characters long and to follow the rules specified by this regular expression `^[a-z][a-z0-9-]{1,61}[a-z0-9]$`. As the uniqueString function generates a string that is 13 characters long, the dnsPrefixString parameter is limited to no more than 50 character

```
"parameters": {
     "dnsPrefixString": {
         "type": "string",
         "maxLength": 50,
         "metadata": {
             "description": "DNS Label for the Public IP. Must be lowercase. It
should match with the following regular expression: ^[a-z][a-z0-9-]{1,61}[a-z0-9]$
or it will raise an error."
         }
     }
 },
 "variables": {
     "dnsPrefix":
"[concat(parameters('dnsPrefixString'),uniquestring(resourceGroup().id))]"
 }
```

✓ When adding a password to a **customScriptExtension**, use the **commandToExecute** property in protectedSettings.

```
"properties": {
     "publisher": "Microsoft.Azure.Extensions",
     "type": "CustomScript",
     "typeHandlerVersion": "2.0",
     "autoUpgradeMinorVersion": true,
     "settings": {
         "fileUris": [
             "[concat(variables('template').assets, '/lamp-app/install_lamp.sh')]"
         ]
     },
     "protectedSettings": {
         "commandToExecute": "[concat('sh install_lamp.sh ',
parameters('mySqlPassword'))]"
     }
 }
```

✓ Any VM Extension should be part of VM resource as a child resource

D. Template Outputs Checklist

✓ If a template creates **publicIPAddresses**, it should have an **outputs** section that returns details of the IP address and the fully qualified domain name. These output values enable you to easily retrieve these details after deployment. When referencing the resource, use the API version that was used to create it.

```
"outputs": {
    "fqdn": {
        "value":
"[reference(resourceId('Microsoft.Network/publicIPAddresses',parameters('publicIPAddr
essName')), '2016-07-01').dnsSettings.fqdn]",
        "type": "string"
    },
```

E. Code Formatting

It is a good practice to pass your template through a JSON validator to remove extraneous commas, parenthesis, brackets that may cause an error during deployment. Try JSONlint or a

linter package for your favourite editing environment (Visual Studio Code, Atom, Sublime Text, Visual Studio, etc.).

It's also a good idea to format your JSON for better readability. You can use a JSON formatter package for your local editor. In Visual Studio, format the document with Ctrl+K, Ctrl+D. In VS Code, use Alt+Shift+F. You can use a JSON formatter package for your local editor or format online using this link.

F. Security Checklist

You should be following below security best practices when designing and developing your template.

- ✓ Network security should be used with all network configuration.
- ✓ NSG shouldn't be configured to allow ANY-ANY ports.
- ✓ It is recommended to ask users to enter public CIDR as parameter template and NSG should be configured to allow access only from the CIDR value.
- ✓ There must not be any hardcoded passwords in template, custom script extensions etc.
- ✓ All artefact's (nested template, scripts etc.) should be stored in azure Quickstart GitHub repo only.

G. Storing Public artifacts Checklist

Most of the times templates may include artifacts and components which needs to be accessed during deployments. Some of these are

- ✓ Custom Scripts to be executed inside a VM
- ✓ PowerShell Modules/DSC Modules to be used
- ✓ Any other resources/components used by deployment

When samples contain scripts, templates or other artifacts that need to be made available during deployment, using the standard parameters for staging those artifacts will enable command line deployment with the scripts provided at the root of the repository. This allows the template to be used in a variety of workflows without changing the templates or default parameters and the artifacts will be staged to a private location, rather than the public GitHub URI.

First, define two standard parameters:

- ✓ **_artifactsLocation**: This is the base URI where all artifacts for the deployment will be staged. The default value should be the samples folder so that the sample can be easily deployed in scenarios where a private location is not required.

- ✓ **_artifactsLocationSasToken**: this is the sasToken required to access _artifactsLocation. The default value should be "" for scenarios where the _artifactsLocation is not secured, for example, the raw GitHub URI.

```
"parameters": {
      "_artifactsLocation": {
          "type": "string",
          "metadata": {
              "description": "The base URI where artifacts required by this template
are located. When the template is deployed using the accompanying scripts, a private
location in the subscription will be used and this value will be automatically
generated."
},
          "defaultValue": "https://raw.githubusercontent.com/Azure/azure-quickstart-
templates/master/201-vm-custom-script-windows/"
      },
      "_artifactsLocationSasToken": {
          "type": "securestring",
          "metadata": {
              "description": "The sasToken required to access _artifactsLocation.
When the template is deployed using the accompanying scripts, a sasToken will be
automatically generated."
          },
          "defaultValue": ""
      }
  },
```

In this example, the custom script extension can be authored using a common pattern that can be applied to all resources that need staged artifacts as well as applied to all samples.

```
"properties": {
      "publisher": "Microsoft.Compute",
      "type": "CustomScriptExtension",
      "typeHandlerVersion": "1.8",
      "autoUpgradeMinorVersion": true,
      "settings": {
      "fileUris": [
          "[concat(parameters('_artifactsLocation'), '/', variables('ScriptFolder'),
'/', variables('ScriptFileName'), parameters('_artifactsLocationSasToken'))]"
        ],
      "commandToExecute": "[concat('powershell -ExecutionPolicy Unrestricted -File
', variables('ScriptFolder'), '/', variables('ScriptFileName'))]"
        }
  }
```

6. Testing and Documentation

A. Testing the Quickstart Template

After developing the solution, you should ensure to test it by deploying in various scenarios such as

- Test template deployment in maximum type subscriptions such as Pay as you go, MSDN, CSP (Cloud Solution Provider), EA (Enterprise Agreement) if possible. This will help in identifying if any type or resources are not available in any of subscription type. For Example, a template involving deployment of Windows 10 Virtual Machines may not work with CSP/EA Subscription as it requires MSDN Subscription. Include findings in README document, if any.

- Test the template deployment using

    - azuredeploy.json based testing via Azure Portal >> New Template Deployment wizard

    - Testing with azuredeploy.json and azuredeploy.parameters.json via Azure PowerShell or Azure CLI.

    - Visual Studio(If being used)

- Ensure the azure portal displays deployment succeed status.

- Test the deployment to understand if deployment is exceeding the default subscription limits(Max number of CPU core etc.) with default parameter values. If yes, include this in README.file pre-requisite section.

- Capture the deployment results and examine time taken by the template to finish

- Examine the dependencies based upon deployment results.

- Perform all post-configuration tasks to verify entire solution end to end flow.

B. Documentation and files

A good Quickstart solution would have following files

- **azuredeploy.json** in template root folder

- **azuredeploy.parameters.json** in template root folder

- **README.md** in template root folder

- **metadata.json** in template root folder

- Any Scripts, artifacts in \scripts\ folder

- Images used in README.md in \images\ folder

- Post deployment guidance document if applicable in images folder.

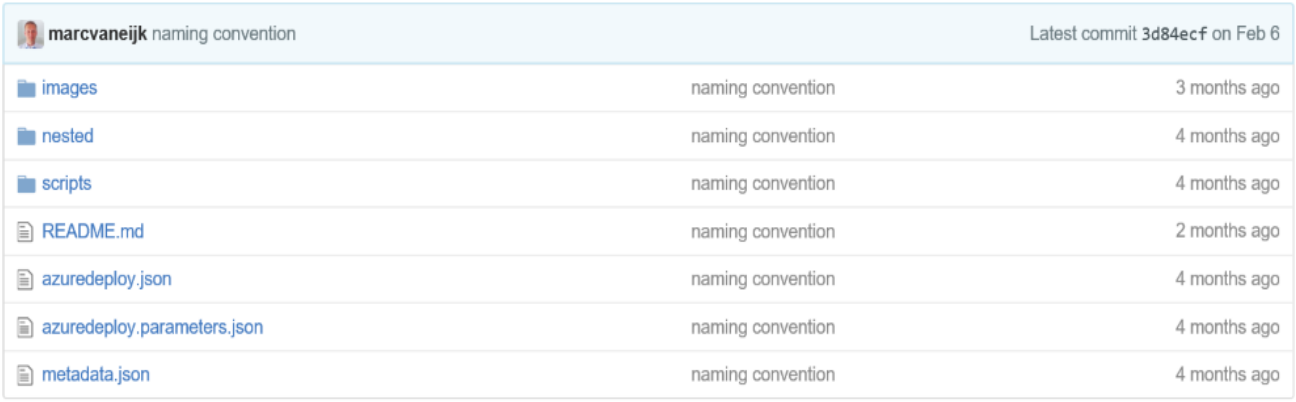
- Pre-deployment guidance document if applicable in images folder.

C. Files and Folder Structure and naming convention

Your Quickstart solution should follow following considerations when it comes to files/folders and names of them.

- Every deployment template and its associated files must be contained in its own **folder**. Name this folder something that describes what your template does. Usually this naming pattern looks like **appName-osName** or **level-platformCapability** (e.g. 101-vm-user-image)
    - **Required** - Numbering should start at 101. 100 is reserved for things that need to be at the top.
    - **Protip** - Try to keep the name of your template folder short so that it fits inside the Github folder name column width.
- Github uses ASCII for ordering files and folder. For consistent ordering **create all files and folders in lowercase**. The only **exception** to this guideline is the **README.md**, that should be in the format **UPPERCASE.lowercase**.
- Include a **README.md** file that explains how the template works. Guidelines below
- The deployment template file must be named **azuredeploy.json**.
- There should be a parameters file named **azuredeploy.parameters.json**. Guidelines below
    - Please fill out the values for the parameters according to rules defined in the template (allowed values etc.), For parameters without rules, a simple "changeme" will do as the acomghbot only checks for syntactic correctness using the ARM Validate Template API.
- The template folder must contain a **metadata.json** file to allow the template to be indexed on azure quickstart page. Guidelines below
- The custom scripts that are needed for successful template execution must be placed in a folder called **scripts**.
- Linked templates must be placed in a folder called **nested**.
- Images used in the README.md must be placed in a folder called **images**.
- Any resources that need to be setup outside the template should be named prefixed with existing (e.g. existingVNET, existingDiagnosticsStorageAccount).

Example below:



- Any post configuration document/guide should be included in pdf format, with link in ReadME file. Pdf files can be places in images folder.

D. README.md document Checklist

The README.md describes your deployment. A good description helps other community members to understand your deployment. The README.md uses Github Flavoured Markdown for formatting text. If you want to add images to your README.md file, store the images in the **images** folder. Reference the images in the README.md with a relative path (e.g. ![alt text](images/namingConvention.png "Files, folders and naming conventions")). This ensures the link will reference the target repository if the source repository is forked. A good README.md contains the following sections

- Azure Partner Quickstart Solution Overview
- Deploy to Azure Button
- Visual Button
- Licenses and Cost related considerations applicable for the quickstart
- Solution Architecture diagram
- Template Deployment Pre-Requisite (Accounts, etc.)
- List of all components deployed
- “How to Deploy” steps
- Post Deployment Steps
- Support Contact Information
- Notes (Any other remarks)
- Terms of use, if any.
- Tags, that can be used for search. Specify the tags comma separated and enclosed between two back-ticks (e.g. Tags: cluster, ha, sql)

Sample README.md file available here

E. Metadata.json file best practices

metadata.json file to allow the template to be indexed on azure quickstart page. A valid metadata.json must adhere to the following structure

```
{
  "itemDisplayName": "",
  "description": "",
  "summary": "",
  "githubUsername": "",
  "dateUpdated": "<e.g. 2015-12-20>"
}
```

The metadata.json file will be validated using these rules

- itemDisplayName : Cannot be more than 60 characters. This would be the title of quickstart
- description
    - This should include detailed overview about the quickstart solution and components used.
    - Cannot be more than 1000 characters
    - Cannot contain HTML.
    - This is used for the template description on the azure quickstart page index template details page
- summary
    - This should include quickstart template high level overview
    - Cannot be more than 200 characters
    - This is shown for template summary on the main azure quickstart page template index page
- githubUsername
    - This is the username of the original template author.
    - This is used to display template author and Github profile pic in the azure quickstart page.
- dateUpdated
    - Date when template was uploaded
    - Must be in yyyy-mm-dd format.
    - The date must not be in the future to the date of the pull request

H. Creating azuredeploy.parameters.json file

You need to create an azuredeploy.parameters.json file for your solution which will have sample value for parameter's used in the templates. A sample parameters file should look like below

```
{
  "$schema": "https://schema.management.azure.com/schemas/2015-01-
01/deploymentParameters.json#",
  "contentVersion": "1.0.0.0",
  "parameters": {
    "adminUsername": {
      "value": "GEN-USER"
    },
    "adminPassword": {
      "value": "GEN-PASSWORD"
    },
    "dnsLabelPrefix": {
      "value": "GEN-UNIQUE"
    }
  }
}
```

Parameters file requires parameter name and value fields only, no other specifications such as parameter data etc should be included here. Do not add parameters which has default value specified main template file(azuredeploy.json).

F. Usage of Logo/Trademarks

Templates development partner has to ensure that any logo's/trademarks which they do not own, or do not have permission to use as per legal terms of respective vendor, must not be used in any README, configuration document, architecture diagram or anywhere else in the quickstart solution.

I. Preparing Template for Automated Travis CI Validation

Microsoft Azure Quickstart repo have automated template validation through Travis CI. These builds can be accessed by clicking the 'Details' link at the bottom of the pull-request dialog. This process will ensure that your template conforms to all the rules mentioned above and will also deploy your template to our test azure subscription.

Parameters File Placeholders

To ensure your template passes, special placeholder values are required when deploying a template, depending what the parameter is used for:

- **GEN-UNIQUE** - use this placeholder for new storage account names, domain names for public ips and other fields that need a unique name. The value will always be alpha numeric value with a length of 18 characters.
- **GEN-UNIQUE-[N]** - use this placeholder for new storage account names, domain names for public ips and other fields that need a unique name. The value will always be alpha numeric value with a length of [N], where [N]can be any number from 3 to 32 inclusive.
- **GEN-SSH-PUB-KEY** - use this placeholder if you need an SSH public key
- **GEN-PASSWORD** - use this placeholder if you need an azure-compatible password for a VM

Here's an example in an azuredeploy.parameters.json file:

```
{
"$schema": "http://schema.management.azure.com/schemas/2015-01-
01/deploymentParameters.json#",
"contentVersion": "1.0.0.0",
"parameters": {
 "newStorageAccountName":{
  "value": "GEN-UNIQUE"
 },
 "adminUsername": {
  "value": "sedouard"
 },
 "sshKeyData": {
  "value": "GEN-SSH-PUB-KEY"
 },
 "dnsNameForPublicIP": {
  "value": "GEN-UNIQUE-13"
 }
}
```

raw.githubusercontent.com Links

If you're making use of **raw.githubusercontent.com** links within your template contribution (within the template file itself or any scripts in your contribution) please ensure the following:

- Ensure any raw.githubusercontent.com links which refer to content within your pull request points to *https://raw.githubusercontent.com/Azure/azure-quickstart-templates/...* and **NOT** your fork.
- All raw.githubusercontent.com links are placed in your azuredeploy.json and you pass the link down into your scripts & linked templates via this top-level template. This ensures we re-link correctly from your pull-request repository and branch.
- Although pull requests with links pointing to *https://raw.githubusercontent.com/Azure/azure-quickstart-templates/master/...* may not exist in the Azure repo at the time of your pull-request, at CI run-time, those links will be converted to *https://raw.githubusercontent.com/{your_user_name}/azure-Quickstart-templates/{your_branch}/....* Be sure to check the casing of *https://raw.githubusercontent.com/Azure/azure-quickstart-templates/master/...* as this is case-sensitive.
- Modify links of images or documents provided in README.md file to use Azure Quickstart repository uri(*https://raw.githubusercontent.com/Azure/azure-quickstart-templates/master/……./images/..)*

If your template has some pre-requisite such as an Azure Active Directory application,service principal, existing virtual network etc. Travis CI Validation doesn't support validation of this. To bypass the CI workflow, include a blank file called **.ci_skip** in the root of your template folder. Travis will skip validating the template once you add this file. You should always update your pull request with justification of adding .**ci_skip** file.

Some of the common reason of Travis CI validation failure are as follows:

- Quickstart needs existing resources such as Virtual Network to be available in subscription for the template deployment to work
- Quickstart needs existing Active Directory Application/SPN to complete the deployment
- Quickstart takes long time(3-4 hours or more) to deploy.
- Quickstart use marketplace items which requires programmatic deployment to be enabled for the item in subscription. You should include this in README file as well.

7. Publishing the Solution.

A. Validation with Template Validation Partner

Template development partner would submit the developed template Microsoft, Microsoft will involve template validation partner to validate the quickstart solution.

Validation Partner would evaluate the Quickstart with respect to below:

- Ensuring all best practices and standards are followed, Quality Assessment.
- Test the end to end Quickstart solution deployment.
- Asses the templates with respect to optimization possibilities.
- Readiness for launch.

Validation partner will share the feedback with development partner, if there's any modification required it has to be accommodated and resubmitted for validation.

Development partner will proceed with submitting the template on Azure GitHub Repo once validation partner approves the solution.

B. Getting ready for Github Pull request

Before creating a pull request in Azure Quickstart GitHub Repo, you need to ensure following

- Verify all folder structure, files naming conventions and other best practices
- Perform a final test
- Replace artifact location Uri's default values, and unique values in parameter file as per instructions provided in Section-5.G
- Modify hyper-links of images used in README.md file to azure quickstart repository url's. More on this here

C. Creating Pull Request in GitHub Azure Quickstart Repo(Using CLI)

Now, you are ready for creating pull request in GitHub for submitting the template. You should have a GitHub account to perform this, if you don't have an existing account, you can get one here

You need to follow these steps to submit a PR.

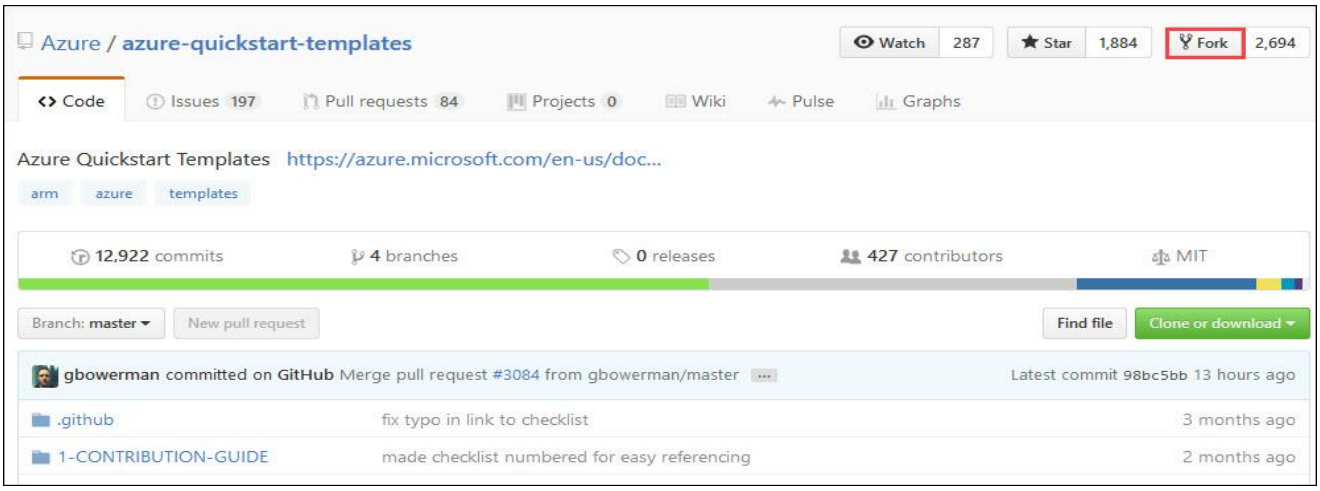
- Access Azure GitHub Repo: https://github.com/Azure/azure-quickstart-templates
- If you don't have Git installed, head over to the official Git Download page and download it. Once downloaded and installed, you might also want to install Posh Git. If you're already using Chocolatey or Windows 10's package manager to install software, you can simply run the following command from an elevated PowerShell (right click, select 'Run as Administrator'):

```
cinst git.install
cinst poshgit

# Restart PowerShell / CMDer before moving on - or run
$env:Path = [System.Environment]::GetEnvironmentVariable("Path","Machine") + ";" +
[System.Environment]::GetEnvironmentVariable("Path","User")

cinst git-credential-winstore
cinst GitHub
```

- Git has multiple ways of pushing - and changed the default behaviour a few years ago. *git config --global push.default* configures what branches will be pushed to the remote. The default in Git 2.0 is simple meaning that whenever git push is run, only the current branch is pushed. Prior to version 2.0, the default behaviour was matching, meaning that a git push would push all branches with a matching branch on the remote.

```
git config --global push.default simple
```

- Fork the repository to your account : Open Azure/azure-Quickstart-templates repository and click the little 'fork' button in the upper right.



- This will create a copy of the repository as it exists in the azure organization (hence called azure/azure-quick-start-templates) in your own account.
- Visit your fork (which should be at github.com/{your_name}/azure-quick-start-templates) and copy the "HTTPS Clone URL". Using this URL, you're able to clone the repository, which is really just a fancy way of saying "download the whole repository, including its history and information about its origin".
- Now that you have Git installed, open up PowerShell. If everything worked correctly, you should be able to run git --version. If that works, navigate to a folder where you'd like to keep the arm-templates repository (and hence, all your arm templates). To get a copy of your fork onto your local machine, run:

```
git clone https://github.com/{YOUR_USERNAME}/azure-quick-start-templates
```

- You can run explorer arm-templates to open up the folder. All the files are there - including the history of the whole repository. The connection to your fork (your_name/arm-templates) is still there. To confirm that, cd into the cloned folder and run the following command, which will show you all the remote repositories registered:

```
git remote -v
```

- The output should be:

```
Users\abc\azure-quick-start-templates [master]> git remote -v
origin https://github.com/sedouard/arm-templates (fetch)
origin https://github.com/sedouard/arm-templates (push)
```

- As you can see, we're connected to your fork of the arm-templates (called "origin"), but currently not connected to the upstream version living in catalystcode/arm-templates. To change that, we can simply add remotes. Run the following command:

```
git remote add upstream https://github.com/azure/azure-quick-start-templates
```

- Entering git remote -v again should give you both repositories - both yours (called "origin") and the one for the whole team (called "upstream").
- Now create a new branch for your template

```
git checkout master
git checkout -b my-new-branch
```

- You can now go ahead and make your changes - adding files, your template, fixing typos. Keep in mind that a branch should host isolated changes: You would normally not fix someone else's template and add a new one in branch. Instead, you create one branch that fixes typos; and another branch for your added template.
- Now that you made your changes, you can "stage" them for a commit. Whenever you stage a file for a commit, you make a snapshot of the file at the time you're staging it for a commit. If you change a file after you staged it, you will have to stage it again. To stage a file, simply run:

```
git add ./path-to/my-file.md
```

- If you just want to stage all files in your current repository (including deletions), run:

```
git add --all .
```

- Now that your changes have been staged, we're ready to commit them. You can either pass the commit command a title for your commit - or omit the parameter, in which case Git will open up the default text editor to create a commit message.
- To commit the quick way:

```
git commit -m "Add template: 3 VMS 1 Subnet"
```

- To commit the long way, allowing you to define both title and message of your commit:

```
git commit
```

- Push your new Branch to Your Fork on GitHub : You wrote a template, made some changes, committed - now we have to make sure that your changes also end up on GitHub. To do so, we have to push your local branch to your fork on GitHub. Run the command below (obviously using the name of the branch you want to push)

```
git checkout NAME_OF_YOUR_BRANCH
```

- Now, head over to the https://github.com/{YOUR_USERNAME}/azure-quick-start-templates repository. In most cases, GitHub will pick up on the fact that you just pushed a branch with some changes to your local fork, assuming that you probably want for those

changes to end up in the upstream branch. To help you, it'll show a little box right above the code asking you if you want to make a pull request.

- Click that button. GitHub will open up the 'Create a Pull Request Page'. It is probably a good idea to notify potential reviewers in your pull request. You can do this by doing an @mention to the maintainer of the repository.
- As soon as you hit the 'Create Pull Request' button, it'll show up in the list of pull requests and trigger some automated validation.
- You can find your pull request on the Github Azure Quickstart repo >> Pull Requests. You should be following this PR thread until your template gets published

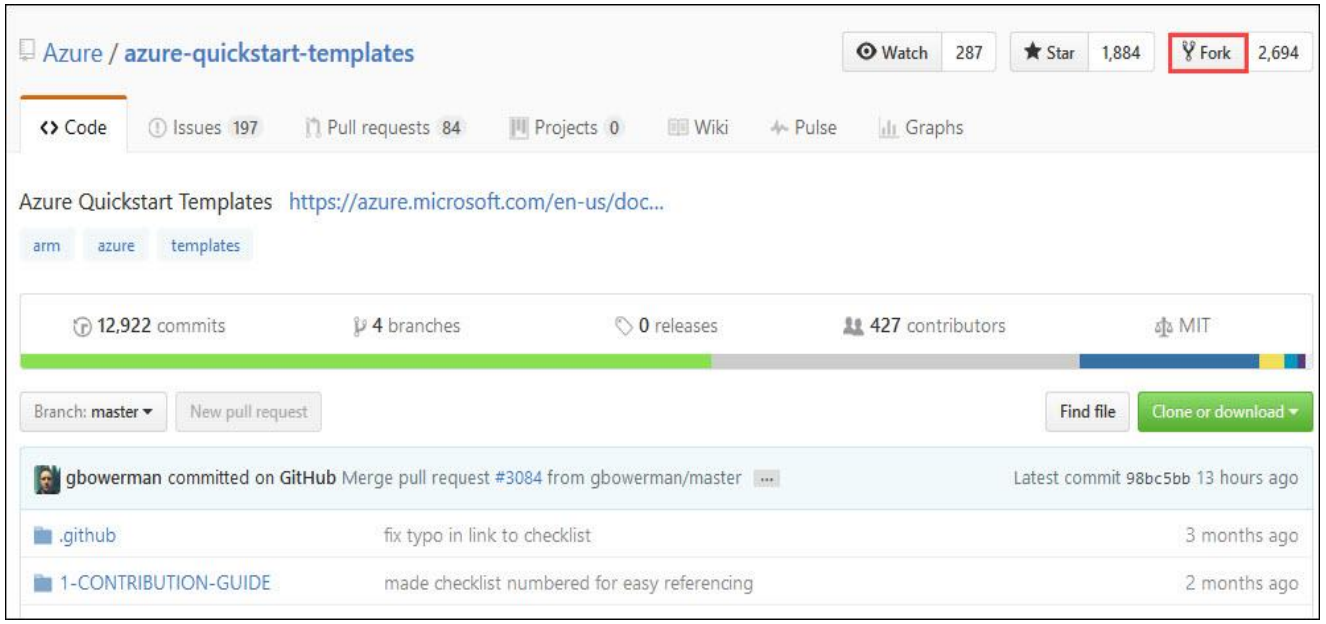
More on GIT commands here.

D. Creating Pull Request in GitHub Azure Quickstart Repo (Using GUI)

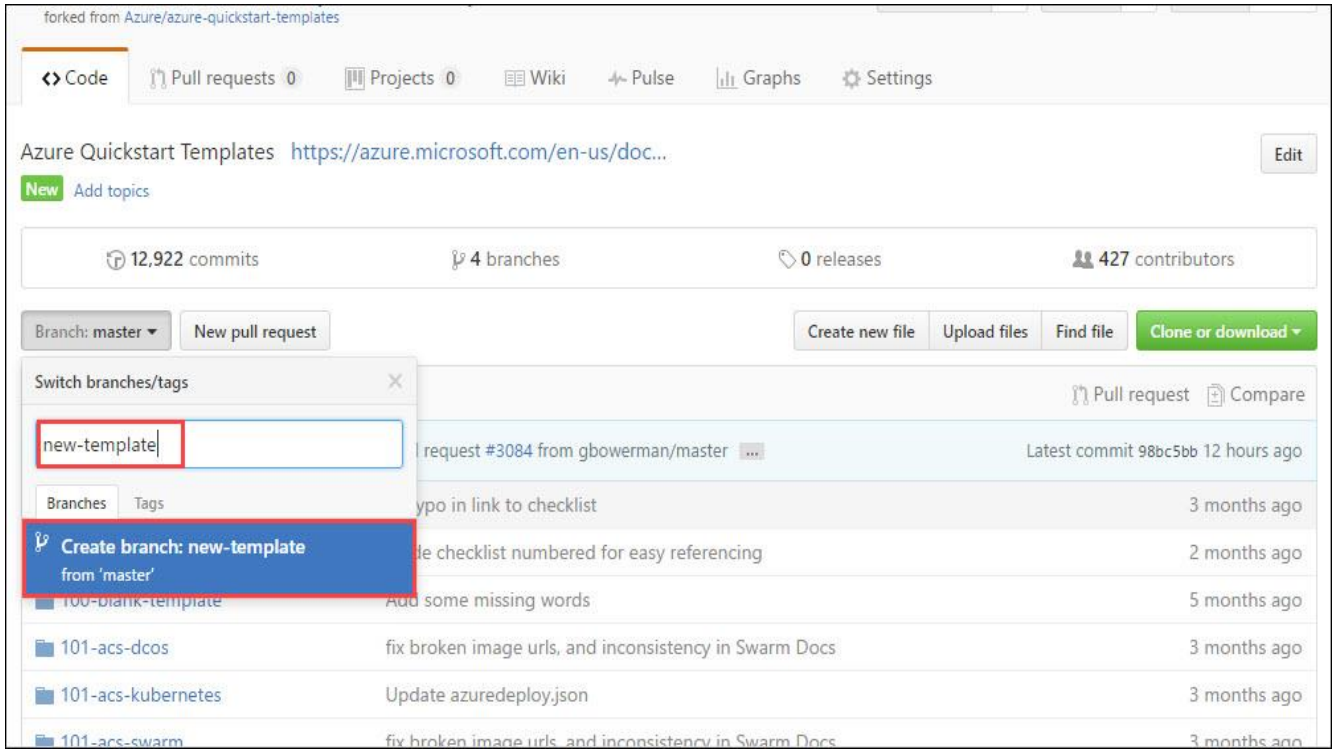
You may follow below process to create pull request in Azure Quickstart GitHub Repo via GUI(Graphical User Interface). You should have a GitHub account to perform this, if you don't have an existing account, you can get one here

Note: Skip this section if you've already created pull request using CLI for your quickstart. This is an alternate method to create PR

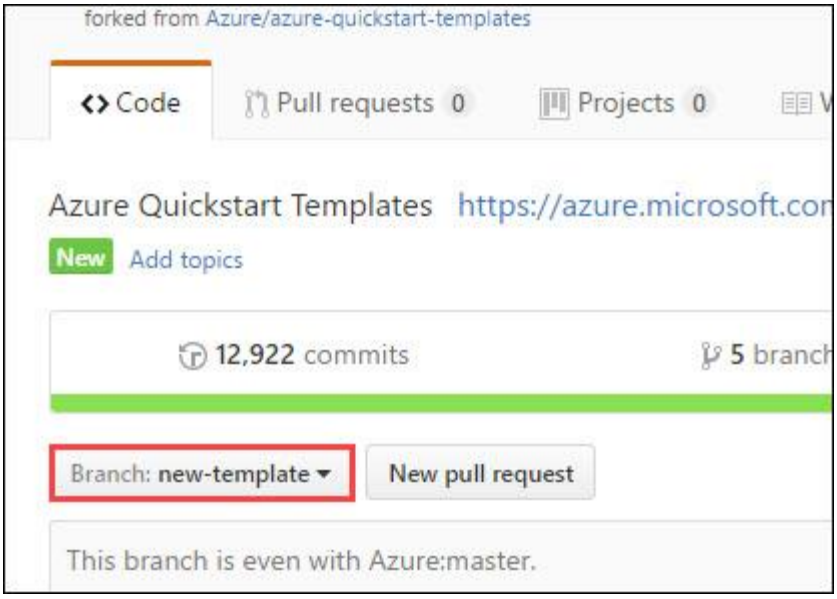
- Login to Github using any Browser
- Access Azure GitHub Repo: https://github.com/Azure/azure-quickstart-templates
- Fork the repository to your account : Open Azure/azure-Quickstart-templates repository and click the little 'fork' button in the upper right.



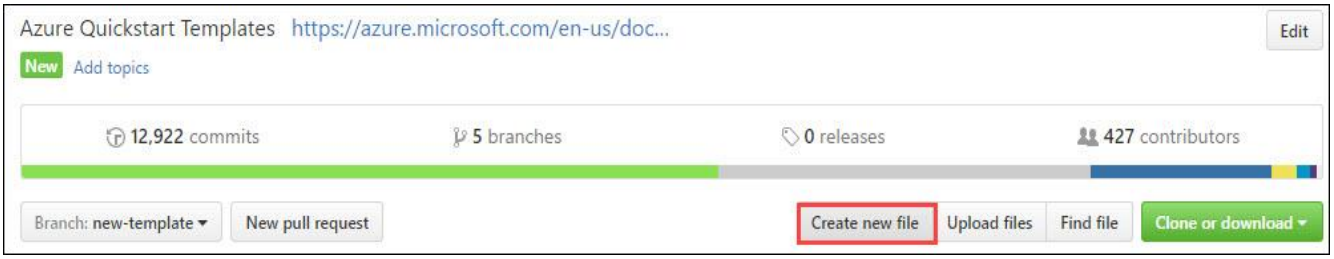
- On where to fork this repository page, select your GitHub account available. Fork process will take 1-2 minutes to finish.
- Once completed, it will redirect you to forked repository in your account.
- Click on **Branch:master** and type new branch name(e.g. your template name-branch) and Click on **Create branch** button.



- You will be redirected to new branch of your repository, verify the branch name.



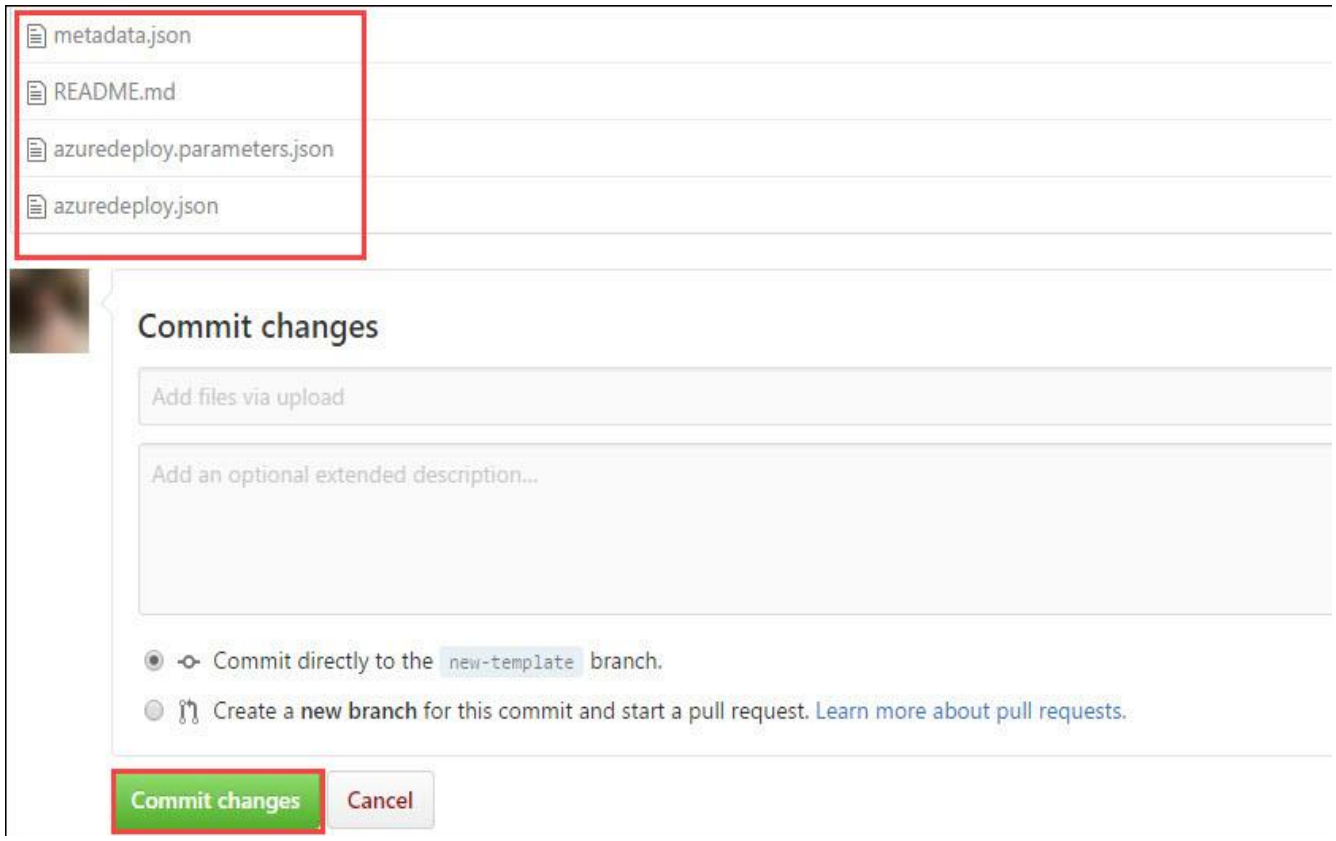
- Click on **create new file** button.



- Enter the template folder name(e.g 101-template-os) and type **/** ,this will treat this as directory name inside root quickstart repo. Inside directory add any temp file as showed in below screenshot.

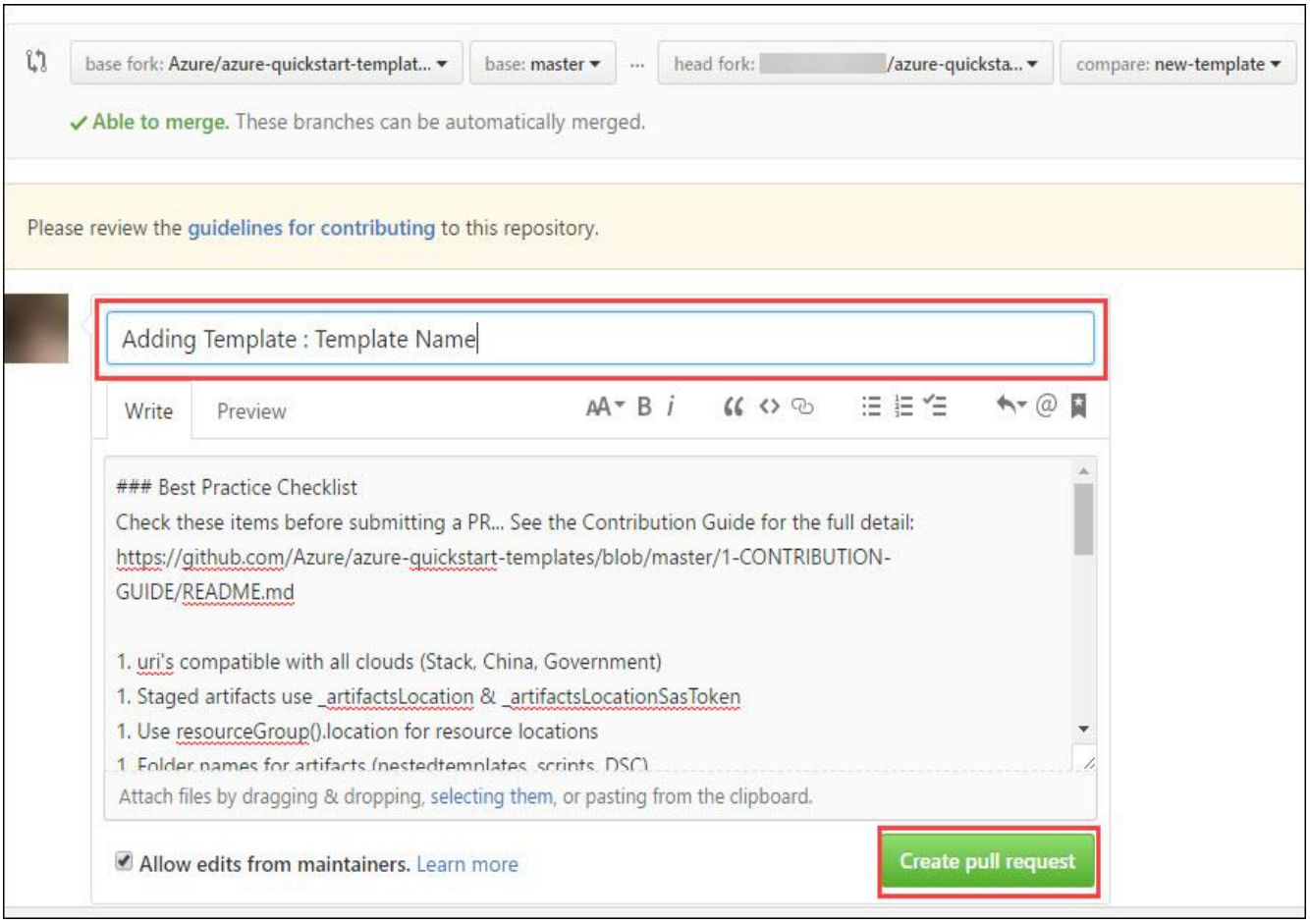
- Click on **Commit Changes** Button
- This will redirect you to your newly created directory. You should see temp file
- Now Click on **Upload Files** and upload all your template file, README.md file and other required template components. You may create a new directory(e.g. scripts , images) by following the same create new file method described in previous step.

- Verify if all files are uploaded, and Click on **Commit Changes.**



- Delete the temp files created earlier while creating the folder for the new template.
- Now head over to your quickstart repo(https://github.com/{YOUR_USERNAME}/azure-quick-start-templates). In most cases, GitHub will pick up on the fact that you just pushed a branch with some changes to your local fork, assuming that you probably want for those changes to end up in the upstream branch. To help you, it'll show a little box right above the code asking you if you want to make a pull request

- Click that button. GitHub will open up the 'Create a Pull Request Page'. It is probably a good idea to notify potential reviewers in your pull request. You can do this by doing an @mention to the maintainer of the repository.



- As soon as you hit the 'Create Pull Request' button, it'll show up in the list of pull requests and trigger some automated validation.
- You can find your pull request on the Github Azure Quickstart repo >> Pull Requests. You should be following this PR thread until your template gets published.

E. Completing Microsoft CLA(Contribution license agreement)

As soon as you submit the PR , **azurecla** test will start and you would be getting notification on PR whether Microsoft CLA is required or not. Microsoft requires you to sign a Contribution License Agreement, which ensure the community is free to use your contributions.

If You have signed cla already using this GitHub account, azurecla bot would automatically update the status as **cla-not-required** otherwise **cla-required** If you haven't signed the CLA. CLA required status will get updated automatically as soon as you complete the CLA signing process.

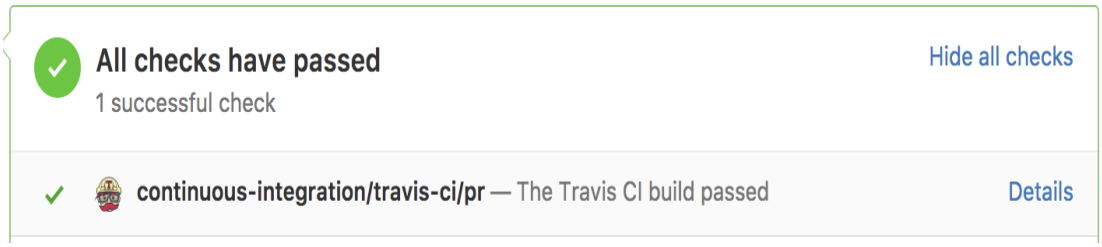
Signing CLA is all electronic and takes few minutes to complete. You can visit this to complete it.

F. Verify Automated Validation Check with Travis

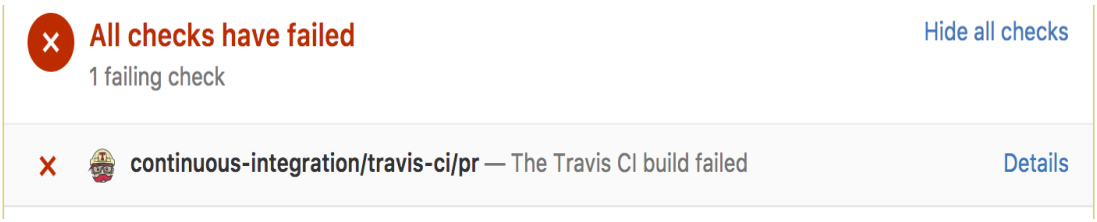
As soon as you submit the PR, automated validation of template using Travis CI should start in next few minutes. You should see the Travis CI validation in progress in your pull request status.

Once Travis CI validation finishes, you should see validation results like below.

- If Travis CI validation completes successfully



- In case of failure



You can click on Details and access Travis logs, few sample logs below. Logs would help you in understanding the reason of failed validation.

Sample Travis Job Logs:

- https://travis-ci.org/Azure/azure-quickstart-templates/builds/108304225
- https://travis-ci.org/Azure/azure-quickstart-templates/builds/159368851

If Travis CLI validation fails, you would do one of the following

- Analyse the error reported by Travis CI, and update your template code with resolution. As soon as you commit change to files in your branch, Travis will start validation again automatically.
- Add **.ci_skip** file if Travis Validation failure is expected due to any of the reason listed here

G. Updating Pull Requests

Once people have reviewed your pull request, it is very likely that you want to update it. Whenever you update the branch in your fork, your pull request will be automatically updated.

Let's look at a scenario: You just pushed your branch (using `git push -u origin NAME_OF_YOUR_BRANCH`), went to GitHub, and created a Pull Request.
To update the PR with additional changes, edit your files in your branch - then, commit the changes and push them to GitHub:

```
git checkout NAME_OF_YOUR_BRANCH
# Make changes, stage all changes for commit
git add --all .
# Commit
git commit
# Push to GitHub
git push git push -u origin NAME_OF_YOUR_BRANCH
```

If you want to update your Pull Request without creating a new commit, you can "amend" your last commit. To do so, call *git commit* with the *--amend* parameter and force-push the result to GitHub, overwriting the previous version.

H. Add Quickstart to QS Portal.

Once PR is merged by Microsoft, QS program support partner will add up the quickstart to Partner QS Portal. Partner will specify all details of QS including title description, architecture diagram link, GitHub Link of template, deployment guides, cost, licenses etc information as applicable.

I. Quickstart Launch

Once QS is Added to portal, Validation partner along with Microsoft will verify the QS one last time for final validation and proof checking, and make the template live on public page of QS portal once approved. Necessary social media updates, blog post etc could follow.

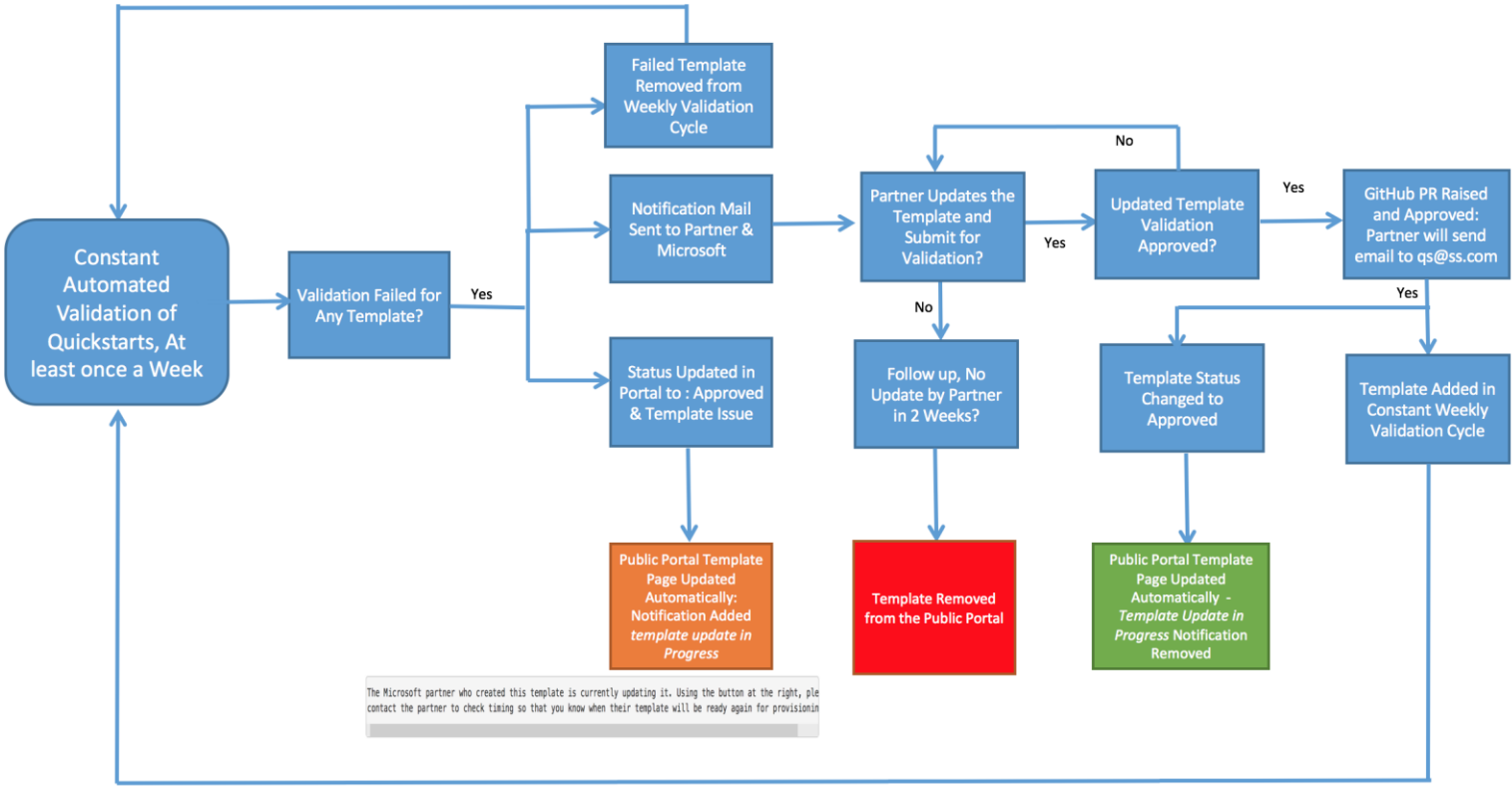
8. Maintenance, Updates and Support

After Quickstart launch, template development partner will need to maintain, update and support the solution as well, as and when required. This would include

- Assisting Customer/partners in deploying the solution as and when required(You would be sharing your support details on Partnerquickstarts portal page)
- Regular Check to ensure the template remains valid and deploys successfully
- Resolution of any issues observed with template in future.
- Update the Quickstart regularly to take advantage of new Azure API, or by adding support for new OS/App versions
- Keep an eye on validation reports on the partnerquickstarts portal.

A. Quickstart Break Fix Process

Quickstart shall be tested regularly to ensure that it remains valid and deploys successfully. All the validation reports are visible under partner quickstart portal. All partner admins shall be notified as soon as any validation record added with failed status. Follow diagram explains the flow when validation fails for any existing valid quickstart.



In a nutshell

- Validation Partner shall keep validating the quickstart regularly and update the status on PQS portal.
- In case of any failure, Validation partner will upload validation result on portal with status as failed. Portal will notify all partner admins and other stakeholders via email about validation failure.
- Status shall be updated to template issue on the portal.
- QS development partner to analyze the failure and take necessary actions to get the QS to working stage.
- Notify Validation partner once QS is back online, Validation partner to verify the QS and make it live on QS Portal.
- QS shall be removed from portal if not valid within 2 weeks of identification. Can be added later one it becomes valid again.

B. Portal Support

Partners can write to quickstartsupport@spektrasystems.com in cas0065 of any questions, issues or any other assistance required related to the partner quickstarts portal.

9. Additional Resources

- Azure Partner Quickstarts portal - https://partnerquickstarts.azurewebsites.net
- Submit quickstart Idea – https://bit.ly/submitapq
- Online Version of Contributor's Guide – https://bit.ly/apqguide
- Quickstart process and portal support email – quickstartsupport@spektrasystems.com
- ARM Templates Documentation: https://docs.microsoft.com/en-us/azure/azure-resource-manager/resource-group-authoring-templates
- GitHub Documentation : https://github.com/Azure/azure-quickstart-templates/blob/master/1-CONTRIBUTION-GUIDE/git-tutorial.md
- Useful Tools https://github.com/Azure/azure-quickstart-templates/blob/master/1-CONTRIBUTION-GUIDE/useful-tools.md
- Azure Quick Start GitHub Repo: https://github.com/Azure/azure-quickstart-templates