SSM Automations

Implementation

Dennis Catharina Johannes Kuijs

June 17, 2025

openremote

Contents

1. Context

This document provides a detailed overview how I implemented the automations on the `AWS` platform using `Amazon Systems Manager` (`SSM`) It also outlines the decisions I made and the challenges encountered throughout the development process.

2. Research based on feedback

After several iterations on the `EBS` data volume implementation, it is now nearly production-ready. I've got one last comment to investigate if it's possible to create an automation to replace an existing `EBS` data volume with an snapshot.

2.1. Investigating SSM Automations

Currently, If an OpenRemote version update fails, we manually rollback the update by replacing the `root` volume with the latest snapshot. This manual process introduces downtime and takes a lot of time.

By automating this process, we can rollback much quicker which results in less downtime and on top of that we can update more frequently.

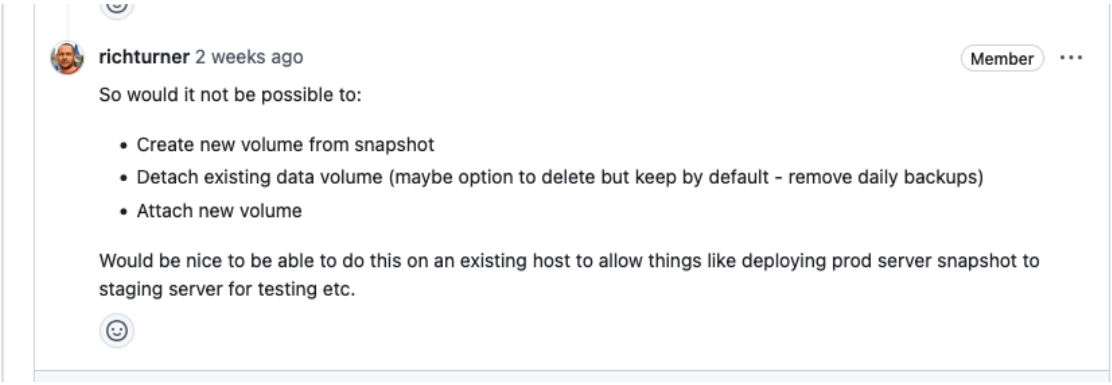

Figure 1: Rich proposed to investigate the possibility to create an SSM document for replacing the current EBS data volume with an existing snapshot

I already created some `SSM` documents for `attaching` , `detaching` , `mounting` and `umounting` the `EBS` data volume. Those documents are used for preparing the `EBS` data volume when provisioning the `EC2` instance for the host.

Despite the fact that these documents are working as expected, I'm not happy in the way the work. These (command) documents are interacting with the AWS API via `CLI` commands in `bash` . Unfortunately, they are executed asynchronously and don't wait for other commands before continuing. As a result of this, I need to implement several loops within the script to wait for a specific status before marking the execution of the command as `success` . I think this approach is to much error prone and not reliable enough to be used in an production environment.

To address this issue, I did some more extensive research on how `Amazon Systems Manager` (`SSM`) is working and which features it offers.

I came across `SSM Automation` , a tool for automating deployment, maintenance and remediation tasks for a variety of AWS Services. After exploring the documentation I found out that with these type of documents it is possible to interact with the AWS API natively using the action `aws:executeAwsApi` . With the action `aws:waitForAwsResourceProperty` you can wait for a variable to become a specific value, for example a `success` status. These automations are not executed asynchrously and wait for each other to complete with an `success` status before continuing. You can even control the next step that is being executed by providing the step name in the `nextStep` property.

This is exactly what we need, all the issues we have with normal `SSM` documents are solved by using `SSM` automations. Since this approach suites the best for our use case, I decided to give it a try by first rewriting the existing documents using `SSM` automation.

3. Rewriting existing SSM documents using SSM automation

3.1. Attach Volume Document

I started by rewriting the attach volume document, this is one of the documents that has the issues that `SSM` automation can solve. The first part of the `SSM` automation documents look almost the same as the `command` documents. Except for the `DocumentType` and `schemaVersion` parameters, `SSM` automation uses `schemaVersion` `0.3` instead of `2.2` . The `DocumentType` must be set to `Automation`

```yaml
SSMAttachVolumeDocument:
  Type: AWS::SSM::Document
  Properties:
    DocumentType: Automation
    DocumentFormat: YAML
    TargetType: /AWS::EC2::Instance
    Name: attach_volume
    Content:
      schemaVersion: '0.3'
      description: 'Script for attaching an EBS data volume'
      parameters:
        VolumeId:
          type: String
          description: '(Required) Specify the VolumeId of the volume that should be attached'
          allowedPattern: '^vol-[a-z0-9]{8,17}$'
        InstanceId:
          type: String
          description: '(Required) Specify the InstanceId of the instance where the volume
↪  should be attached'
          allowedPattern: '^i-[a-z0-9]{8,17}$'
        DeviceName:
          type: String
          description: '(Required) Specify the Device name where the volume should be mounted
↪  on'
          allowedPattern: '^/dev/sd[b-z]$'
```

The parameters are described the exact same way, no changes need to be made here. This document have 3 different parameters:

- `VolumeId` : The `EBS` data volume that needs to be attached to the instance.
- `InstanceId` : The `EC2` instance where the `EBS` data volume needs to be attached.
- `DeviceName` : The `DeviceName` where the volume needs to be mounted on.

After the initial configuration, I started describing the different steps that need to be excuted in the `mainSteps` block. Instead of using the `aws:runShellScript` action and specify the commands in `bash` that are being executed on the targeted `EC2` machine, I specify the different AWS `API` actions by using the `aws:executeAwsApi` action.

3.1.1. Get Instance Details

The first step is to retrieve the `EC2` instance details using the `InstanceId` that is specified in the `parameters` section. This is handeled by the `DescribeInstances` API.

When the instance details are fetched, The `hostname` is retrieved by filtering the `tags` property and searching for the tag with the `Key==Name` . This value is pushed to the `outputs` section so it becomes available for the next steps. The `hostname` variable is very important in this process as it is used in different places to reference the various `AWS` components, for example, to target the correct volume in the `DLM` policy for creating `snapshots` .

```
# Retrieve instance details to get the Host name.
- name: GetInstanceDetails
  action: aws:executeAwsApi
  timeoutSeconds: 120
  onFailure: Abort
  inputs:
   Service: ec2
   Api: DescribeInstances
   InstanceIds:
    - '{{ InstanceId }}'
  outputs:
    - Name: Host
      Selector: $.Reservations[0].Instances[0].Tags[?(@.Key == 'Name')].Value
      Type: String
   nextStep: AttachVolume
```

After this step is successfully executed, the automation will continue by executing the step provided in the `nextStep` parameter. In this case the `AttachVolume` step.

3.1.2. Attach Volume

After successfully retrieving the `hostname` , the `EBS` data volume can be attached to the instance specified in the `parameters` section. The `EBS` data volume will be mounted on the specified `Devicename` , this can be any value between `/dev/sd[b]-[z]` as long as the `Devicename` is not already in used by another volume. The `/dev/sda` is reserved for the `root` device and cannot be used.

Because the next steps are dependent on this step, I added the `isCritical=true` parameter to ensure that the automation will only continue when this step is executed successfully. When the step failed, it automatically stops running the automation.

```yaml
# Attach EBS data volume to specified instance
- name: AttachVolume
  action: aws:executeAwsApi
  timeoutSeconds: 120
  onFailure: Abort
  isCritical: true
  inputs:
    Service: ec2
    Api: AttachVolume
    VolumeId: '{{ VolumeId }}'
    InstanceId: '{{ InstanceId }}'
    Device: '{{ DeviceName }}'
  nextStep: WaitForVolumeAttachment
```

When the `EBS` data volume is successfully attached to the instance, the automation will continue by executing the `WaitForVolumeAttachment` step.

3.1.3. Wait For Volume Attachment

Once the `EBS` data volume is attached to the instance, it's important to check that the volume is correctly attached to the instance (`state=attached`). In the previous `SSM` command documents, I checked the status every 30 seconds by executing an `CLI` command in a `bash` while loop. With `SSM` automation, I can use the `aws:waitForAwsResourceProperty` action and check for an `AWS` resource to become a specific value. The step will wait until the value specified in the `DesiredValues` is retrieved.

In this case, I used the `DescribeVolumes` API to retrieve the volume details using the `VolumeId` specified in the `parameters` section. With the `PropertySelector` I target the volume's `state` parameter from the API `Response` object using the `JSONPath` notation. With the `DesiredValues` I can specify the exact value that the parameter retrieved from the `PropertySelector` must have before the step is marked as `successfull` and continuing to the next step provided in the `nextStep` parameter.

When something goes wrong, the step will not receive the value specified in the `DesiredValues` parameter which eventually can lead to a infinite loop of waiting. Therefore, It's recommend to configure a timeout when using the `aws:waitForAwsResourceProperty` by providing the `timeoutSeconds` parameter. When no specific timeout is specified the default of 1 hour (`3600` seconds) will be used.

```yaml
# Wait until the EBS data volume is successfully attached
- name: WaitForVolumeAttachment
  action: aws:waitForAwsResourceProperty
  timeoutSeconds: 120
  onFailure: Abort
  inputs:
```

```
  Service: ec2
  Api: DescribeVolumes
  VolumeIds:
    - '{{ VolumeId }}'
  PropertySelector: '$.Volumes[0].Attachments[0].State'
  DesiredValues:
    - attached
  nextStep: MountVolume
```

When the volume is in the `attached` state, the automation will continue by executing the `MountVolume` step.

3.1.4. Mount Volume

After successfully attaching the volume to the `EC2` instance it needs to be mounted to a `directory` before it can be used. After investigating the different `SSM` automation actions it turns out that executing `bash` commands directly on the `EC2` instance is currently not supported by `SSM` automations. The action `aws:executeScript` only allows `python` and `powershell` scripts to be executed within the automation. These scripts are used for processing more complex logic, but cannot run on `EC2` instances.

To mount the volume to the instance you must execute several `linux` commands directly on the `EC2` instance. To achieve this, I decided to use the existing `mount_volume` document that uses the `DocumentType` command. This document has support for running commands and scripts on the targeted `EC2` instance by using the `aws:runShellScript` action.

To execute the `mount_volume` document within the `SSM` automation, I use the `aws:runCommand` action. I specify the document that needs to be executed by configuring the `DocumentName` parameter. Parameters that can be configured within the document can be specified in the `Parameters` section by providing each individual parameter by its name. For mounting the volume there are 2 parameters required:

- `DeviceName` : The `DeviceName` where the volume needs to be mounted on.
- `InstanceIds` : The `EC2` instance where the `EBS` data volume needs to be attached.

The parameters for the `mount_volume` document are populated with the parameter values that are configured at the top of the automation document. They are filled in by the user when executing the automation and will be passed to the other documents when needed.

```
# Mount EBS data volume to specified instance
- name: MountVolume
  action: aws:runCommand
```

```
  timeoutSeconds: 120
  onFailure: Abort
  isCritical: true
  inputs:
   DocumentName: mount_volume
   Parameters:
    DeviceName:
        - '{{ DeviceName }}'
    InstanceIds:
        - '{{ InstanceId }}'
   nextStep: WaitForVolumeMounting
```

When the `mount_volume` document is successfully executed, it is important to check if the execution is succeeded. Therefore, in the next step, I will check for the execution status.

3.1.5. Wait For Volume Mounting

Before continuing to the next step, It is important to check if the `mount_volume` document has successfully executed all the commands that are described. In this step, I will use the `aws:waitForAwsResourceProperty` again to check for a specific value. I used the `GetCommandInvocation` API to retrieve the invocation details from the `mount_volume` document by specifing the `CommandId` parameter with the `CommandId` from the previous step.

In the `PropertySelector` , I target the `StatusDetails` parameter from the API `Response` object to check for the document's execution status. In the `DesiredValues` parameter I specify that the value must be `Success` before moving to the next step.

```
# Wait until the EBS data volume is successfully mounted
- name: WaitForVolumeMounting
  action: aws:waitForAwsResourceProperty
  timeoutSeconds: 120
  onFailure: Abort
  inputs:
   Service: ssm
   Api: GetCommandInvocation
   CommandId: '{{ MountVolume.CommandId }}'
   InstanceId: '{{ InstanceId }}'
   PropertySelector: '$.StatusDetails'
   DesiredValues:
     - Success
   nextStep: UpdateTag
```

When the `mount_volume` document is successfully executed, the automation will continue to the next step. At this point, the volume is successfully attached and mounted to the targeted `EC2` instance.

3.1.6. Update Tag

The last step in this automation is updating one of the `tags` that are configured on the `EBS` data volume. When the `EBS` data volume is provisioned using the `provision host` CI/CD workflow, it creates an `DLM` policy for automatic snapshot creation. Amazon Data Lifecycle manager uses `tags` to target the `volumes` or `instances` that should be covered by the policy.

In the previous example, I configured the policy to look for the `Name` tag with the value `{Hostname}/data`, this setup was working fine, but is hard to maintain. When the `Name` tag gets updated, the name in the `AWS` management portal also changed, which makes it hard to identity the different volumes. On top of that, when `detaching` the volume, it should not be targeted by the `DLM` policy anymore. To achieve this, the tag needs to be updated and the `Name` tag is not suitable for this.

Therefore, I created a custom tag with the name `Type`. When creating the `EBS` data volume during the `provision host` CI/CD workflow, it will configure the tag with the value `or-data-not-in-use`. When the volume is attached to the instance, the tag will be updated with the value `or-data-in-use`. The `DLM` policy is configured to target only `volumes` with this tag. When the volume is detached from the `EC2` instance, the tag will be updated to `or-data-not-in-use` and therefore not be targeted by the policy anymore.

To make this approach more granular, I added the `hostname` in front of the `tag` value to make sure that the `DLM` policy is only targetting the volumes from a specific host. Otherwise, when multiple `hosts` are deployed within the same `AWS` account, multiple `DLM` policies are targetting the same `volumes`. This could result in multiple `snapshot` creations at the same time.

When the previous steps are successfully executed, the `UpdateTag` step will update the `tag` from the `EBS` data volume that is specified in the `ResourceIds` parameter to `or-data-in-use`. This ensures that the volume is targeted by the `DLM` policy and automatic snapshots are being created.

```yaml
# Change tag to ensure the EBS data volume is targeted by the DLM policy
- name: UpdateTag
  action: aws:createTags
  timeoutSeconds: 120
  onFailure: Abort
  inputs:
   ResourceType: EC2
   ResourceIds:
    - '{{ VolumeId }}'
   Tags:
    - Key: Type
      Value: '{{ GetInstanceDetails.Host }}-or-data-in-use'
```

3.2. Detach Volume Document

After rewriting the `attach_volume` document, I tested it and confirmed the solution was working as expected. Using `SSM` automation, the problems and bottlenecks that occur in the previous approach were solved and the solution is running stable with less problems. Becuase of the success, I decided to rewrite the `detach_volume` document as well.

The document uses the same `DocumentType` and `schemaVersion` as the previous automation. For detaching the volume, only an `VolumeId` is required. This is configured within the `parameters` section and can be filled in before running the automation.

```
Type: AWS::SSM::Document
Properties:
  DocumentType: Automation
  DocumentFormat: YAML
  TargetType: /AWS::EC2::Instance
  Name: detach_volume
  Content:
    schemaVersion: '0.3'
    description: 'Script for detaching an EBS data volume'
    parameters:
      VolumeId:
        type: String
        description: '(Required) Specify the VolumeId of the volume that should be detached'
        allowedPattern: '^vol-[a-z0-9]{8,17}$'
```

3.2.1. Get Volume Details

To make this document as simple as possible and to prevent errors because of wrong configuration, I decided to retrieve the `InstanceId` and `DeviceName` using the `VolumeId` specified in the `parameters` section at the top of this document. This ensures that always the correct `Instance` and `DeviceName` associated with the volume is retrieved instead of manually entering them.

The volume details are retrieved using the `DescribeVolumes` API by specifying the `VolumeId` from the `parameters` section at the top. After successfully fetching the volume details the `DeviceName` and `InstanceId` parameters can be retrieved by targeting the parameters from the API `Response` object using the `JSONPath` notation. The parameters are then pushed to the `outputs` section so they can be used in the next steps.

```
# Retrieve EBS data volume details to get the DeviceName and InstanceId
- name: GetVolumeDetails
  action: aws:executeAwsApi
  timeoutSeconds: 120
  onFailure: Abort
```

```
  inputs:
    Service: ec2
    Api: DescribeVolumes
    VolumeIds:
      - '{{ VolumeId }}'
    outputs:
      - Name: DeviceName
        Selector: '$.Volumes[0].Attachments[0].Device'
      - Name: InstanceId
        Selector: '$.Volumes[0].Attachments[0].InstanceId'
  nextStep: GetInstanceDetails
```

After the volume details are successfully retrieved, the automation will continue to the next step.

3.2.2. Get Instance Details

The same as in the `attach_volume` automation, I retrieve the instance details using the `DescribeInstances` API and the `InstanceId` that is retrieved in the `GetVolumeDetails` step. To get the `hostname` , I used the `JSONPath` notation to filter the API `Response` object and look for the parameter `tags` filtered by name (`Key==Name`). This value is pushed to the `outputs` section so it can be used by the next steps.

```
# Retrieve instance details to get the Hostname
- name: GetInstanceDetails
  action: aws:executeAwsApi
  timeoutSeconds: 120
  onFailure: Abort
  inputs:
    Service: ec2
    Api: DescribeInstances
    InstanceIds:
      - '{{ GetVolumeDetails.InstanceId }}'
    outputs:
      - Name: Host
        Selector: $.Reservations[0].Instances[0].Tags[?(@.Key == 'Name')].Value
        Type: String
    nextStep: UmountVolume
```

After the instance details are successfully retrieved, the automation will continue by executing the `UmountVolume` step.

3.2.3. Umount Volume

Before the volume can be detached from the `EC2` instance, it is recommended to first `umount` the volume to prevent any unexpected system behaviour or unrecoverable `I/O` errors with the volume. To

`umount` the volume, I need to run several `linux` commands on the `EC2` machine itself. Therefore, I use the same `command` document as the `mount` volume.

This step uses the `aws:runCommand` action to execute the document `umount_volume` with the `DeviceName` parameter. This value is retrieved from the `GetVolumeDetails` step and will be used to `umount` the correct volume. The `InstanceIds` parameter specifies the `InstanceId` where this document needs to be executed.

```
# Execute the SSM command that umounts the specified EBS data volume
- name: UmountVolume
  action: aws:runCommand
  timeoutSeconds: 120
  onFailure: Abort
  inputs:
    DocumentName: umount_volume
    Parameters:
      DeviceName:
        - '{{ GetVolumeDetails.DeviceName }}'
    InstanceIds:
      - '{{ GetVolumeDetails.InstanceId }}'
  nextStep: WaitForUmount
```

When the `umount_volume` document is successfully executed, the automation will move forward to the `WaitForUmount` step.

3.2.4. Wait For Umount

Before detaching the volume, it is important to check if all the steps within the `umount_volume` document are properly executed. In this step, I will check the execution status by using the `aws:waitForAwsResourceProperty` action. With the `GetCommandInvocation` API, I retrieve the command invocation details using the `CommandId` from the `UmountVolume` step. Then, I used the `PropertySelector` to filter the `StatusDetails` parameter from the API `Response` object.

```
# Wait until the SSM document for umounting the EBS data volume is successfully executed
- name: WaitForUmount
  action: aws:waitForAwsResourceProperty
  timeoutSeconds: 120
  onFailure: Abort
  inputs:
    Service: ssm
    Api: GetCommandInvocation
    CommandId: '{{ UmountVolume.CommandId }}'
    InstanceId: '{{ GetVolumeDetails.InstanceId }}'
    PropertySelector: '$.StatusDetails'
    DesiredValues:
      - Success
  nextStep: DetachVolume
```

When the status becomes `Success`, the value that is configured in the `DesiredValues` section, the automation will continue to the `DetachVolume` step.

3.2.5. Detach Volume

The next step in the process is to detach the volume from the `EC2` machine. When the `umount_volume` document is successfully executed, the volume is properly umounted and can safely be removed from the virtual machine. In this step, I use the `DetachVolume` API to detach the volume using the `VolumeId` that is specified in the parameters section at the top of this document.

```yaml
# Detach the EBS data volume
- name: DetachVolume
  action: aws:executeAwsApi
  timeoutSeconds: 120
  onFailure: Abort
  isCritical: true
  inputs:
    Service: ec2
    Api: DetachVolume
    VolumeId: '{{ VolumeId }}'
  nextStep: WaitForVolumeDetachment
```

When this step is successfully executed, the step continues to the `WaitForVolumeDetachment` step to check if the volume is successfully detached.

3.2.6. Wait For Volume Detachment

Before moving to the last step of this automation, it is important to check if the volume is successfully detached from the `EC2` machine. In this step, I use the `aws:waitForAwsResourceProperty` action to check the status of the volume with the `DescribeVolumes` API. I use the `PropertySelector` to grab the volume's `state` from the API `Response` object using the `JSONPath` notation. When the state is `available`, the volume is successfully detached and the step will be marked as completed.

```yaml
# Wait until the EBS data volume is succesfully detached
- name: WaitForVolumeDetachment
  action: aws:waitForAwsResourceProperty
  timeoutSeconds: 120
  onFailure: Abort
  inputs:
    Service: ec2
    Api: DescribeVolumes
    VolumeIds:
      - '{{ VolumeId }}'
```

```
    PropertySelector: '$.Volumes[0].State'
    DesiredValues:
      - available
  nextStep: UpdateTag
```

The automation will continue to the `UpdateTag` step when it detects that the volume is successfully detached from the `EC2` instance.

3.2.7. Update Tag

When the volume is detached from the `EC2` machine, it's important to update the `type` tag to ensure that the volume is no longer targeted by the `DLM` policy. I use the `aws:createTags` action to update the tag from the volume that is targeted in the `ResourceIds` parameter to `or-data-not-in-use`. The `hostname` is included to make this `tag` unique when multiple hosts are deployed within the same `AWS` account. Otherwise multiple `DLM` policies would target the same volume which results in duplicate snapshots.

```
# Change tag to ensure the volume is no longer targeted by the DLM policy
- name: UpdateTag
  action: aws:createTags
  timeoutSeconds: 120
  onFailure: Abort
  inputs:
    ResourceType: EC2
    ResourceIds:
      - '{{ VolumeId }}'
    Tags:
      - Key: Type
    Value: '{{ GetInstanceDetails.Host }}-or-data-not-in-use'
```

When this step is executed successful, the `detach_volume` automation is completed.

3.3. Mount Volume Document

To mount the `EBS` data volume to the `EC2` instance, I used the previous created `SSM` command document. This document is executing the following steps on the `EC2` instance:

- `Mount Volume` - This script checks for an existing `filesystem` on the volume and based on that creates a new one or mounts the existing.
- `Add File System Entry` - To ensure that the volume is automatically mounted on instance restart, I created a script that adds an entry in the `/etc/fstab` file.

- `Start Docker` - After the volume is mounted, the docker `service` and `socket` are started. When there are existing `Docker` containers on the system, they are automatically started during this step.

The `mount_volume` document uses the `DocumentType` command and `schemaVersion` 2.2. The only parameter that is required is the `DeviceName` where the volume needs to be mounted on.

```
Type: AWS::SSM::Document
Properties:
  DocumentType: Command
  DocumentFormat: YAML
  TargetType: /AWS::EC2::Instance
  Name: mount_volume
    Content:
      schemaVersion: '2.2'
      description: 'Script for mounting an EBS data volume'
      parameters:
        DeviceName:
          type: String
          description: '(Required) Specify the Device name where the volume should be mounted
↪  on'
          allowedPattern: '^/dev/sd[b-z]$'
```

3.3.1. Mount Volume

Before the volume can be mounted to the `EC2` instance, it is important to check if the volume has an existing filesystem. Creating a new filesystem on top of an existing filesystem will overwrite the data. To check for an existing filesystem, I use the `blkid` command and filter on `TYPE` using the `DeviceName` provided in the parameters section at the top of the document.

When there is no existing filesystem, the script will create one using the `xfs` file system and mountes the `/var/lib/docker/volumes` directory on the `DeviceName` specified. If there is already an filesystem available, for example when an volume is created from an existing snapshot, the system will only mount the volume.

```
# Mount the specified EBS data volume to the instance
- name: MountVolume
  action: aws:runShellScript
  inputs:
    runCommand:
      - |
        FILESYSTEM=$(blkid -o value -s TYPE {{ DeviceName }})
        if [ -z "$FILESYSTEM" ]; then
          mkfs -t xfs {{ DeviceName }}
          mount {{ DeviceName }} /var/lib/docker/volumes
        else
```

```
          mount {{ DeviceName }} /var/lib/docker/volumes
        fi
```

3.3.2. Add File System Entry

To ensure the volume is automatically mounted on instance reboot, an entry in the file systems table must be created in the `/etc/fstab` file. The script first fetches the volume's `UUID` . The `UUID` is used to target the volume within the file systems table. It's possible to use the `DeviceName` here instead, but this isn't recommended as the `DeviceName` may change. The `UUID` is persistent throughout the life of the volume. Even when you restore the volume from an snapshot.

After retrieving the `UUID` , the script first makes a backup of the `/etc/fstab` in case something goes wrong. When the file systems table is corrupt the instance may not be able to reboot anymore. If the backup is succesful created, the script will update the file systems table by adding a new line to the `/etc/fstab` file. The entry consists of the following components:

- `UUID` - The volume's `UUID` retrieved from the `blkid` command
- `Directory` - The directory that needs to be mounted (`/var/lib/docker/volumes`)
- `File System` - The file system type (`xfs`)
- `Tags` - Different tags to control the behaviour of the volume. the `nofail` tag ensures that the instance can boot even if the volume cannot be mounted. The `defaults` tag configures several standard settings to the volume.
- `Options` - Different options to control the behaviour of the volume, the `0` means the volume isn't backupped by the `dump` command, the `2` means that the volume isn't an `root` device.

Updating this file is very important. Therefore, the script will throw an error if something goes wrong.

```
# Add the specified EBS data volume to the file system table
- name: AddFileSystemEntry
  action: aws:runShellScript
  inputs:
    runCommand:
      - |
        UUID=$(blkid -o value -s UUID {{ DeviceName }})
        if [ -n "$UUID" ]; then
          cp /etc/fstab /etc/fstab.orig
          echo "UUID=$UUID /var/lib/docker/volumes xfs defaults,nofail 0 2" >> /etc/fstab
        else
          echo "Failed to add /etc/fstab entry .. UUID is not found"
          exit 1
        fi
```

3.3.3. Start Docker

When the file systems table is successfully updated, the script will start the `Docker` `socket` and `service` again using the `systemctl` command. Existing containers will automatically try to start again. For extra safety, both the `socket` and `service` are disabled when the volume gets detached. This prevents the `Docker` service accidentally starts when the updating process is in progress.

```
# Start the Docker service and socket
- name: StartDocker
  action: aws:runShellScript
  inputs:
    runCommand:
      - systemctl start docker.socket docker.service
```

3.4. Umount Volume

To umount the `EBS` data volume from the `EC2` instance, the `SSM` document that was previously created is being used. This document looks almost identical to the `mount_volume` document. The following steps are being executed:

- `Umount Volume` - This script checks for an existing `filesystem` on the volume and based on that creates a new one or mounts the existing.
- `Remove File System Entry` - To ensure that the volume is automatically mounted on instance restart, I created a script that adds an entry in the `/etc/fstab` file.
- `Stop Docker` - After the volume is mounted, the docker `service` and `socket` are started. Existing `Docker` containers are automatically started during this step.

The `umount_volume` document uses the `DocumentType` command and `schemaVersion` 2.2. The only parameter that is required is the `DeviceName` that needs to be umounted

```
Type: AWS::SSM::Document
Properties:
  DocumentType: Command
  DocumentFormat: YAML
  TargetType: /AWS::EC2::Instance
  Name: umount_volume
    Content:
      schemaVersion: '2.2'
      description: 'Script for umounting an EBS data volume'
      parameters:
        DeviceName:
          type: String
          description: '(Required) Specify the Device name where the volume is mounted on'
          allowedPattern: '^/dev/sd[b-z]$'
```

3.4.1. Stop Docker

The first step in this process is to stop the `Docker` `service` and `socket` using the `systemctl` command to prevent any issues with the running containers that are using the `EBS` data volume.

```
# Stop the Docker service and socket
- name: StopDocker
  action: aws:runShellScript
  inputs:
    runCommand:
      - systemctl stop docker.socket docker.service
```

3.4.2. Remove File System Entry

When the `docker` `service` and `socket` are succesfully stopped all the `docker` containers are shutdown. After that, the script will remove the `EBS` data volume from the file systems table using the `sed` command. This ensures that the instance wouldn't try to mount an non-attached volume the instance. The script will create a backup first in case the file becomes corrupt.

To delete the correct entry in the `/etc/fstab` file, the `sed` command uses the volume's `UUID` to identify the row that needs to be removed. If the `UUID` cannot be retrieved using the volume's `DeviceName` , the system will throw an error and the operation failes.

```
# Remove the specified EBS data volume from the file systems table
- name: RemoveFileSystemEntry
  action: aws:runShellScript
  inputs:
  runCommand:
    - |
      UUID=$(blkid -o value -s UUID {{ DeviceName }})
      if [ -n "$UUID" ]; then
        cp /etc/fstab /etc/fstab.orig
        sed -i '/UUID='$UUID'/d' /etc/fstab
      else
        echo "Failed to remove /etc/fstab entry .. UUID is not found"
        exit 1
      fi
```

3.4.3. Umount Volume

After the entry is successfully removed from the `/etc/fstab` file, the `EBS` data will be umounted from the `EC2` instance. Umounting the volume is not required, but acts as an safeguard to prevent any unexepected errors with the volume. The script uses the `findmnt` command to identify if the

volume is actually mounted. Trying to `umount` an volume that isn't actually mounted results in an error. Therefore, this check is neccesary.

If the device is not mounted, the system will skip this step and `echo` a message for reference. Otherwise, the volume will be unmounted from the `EC2` instance.

```yaml
# Umount the specified EBS data volume
- name: UmountVolume
  action: aws:runShellScript
  inputs:
  runCommand:
    - |
      MOUNT=$(findmnt -S {{ DeviceName }})
      if [ -n "$MOUNT" ]; then
          umount {{ DeviceName }}
      else
          echo "Device not mounted .. Skipping step"
      fi
```

4. Replace snapshot automation

After rewriting all the `SSM` documents to the new `SSM` automation method, I developed a new `SSM` automation to replace an existing `EBS` data volume with an snapshot. This task is currently performed manually and takes a lot of time. By automating this, the rollback process is much faster, more reliable and less error prone.

The document uses the same `DocumentType` and `schemaVersion` as the other documents. For this automation the following parameters are required:

- `VolumeId` - To specify which `EBS` data volume needs to be replaced with the snapshot.
- `SnapshotId` - To specify which snapshot should be used for creating the new `EBS` data volume.
- `DeleteVolume` - To specify if the current `EBS` data volume should be kept or deleted.

```
Type: AWS::SSM::Document
Properties:
  DocumentType: Automation
  DocumentFormat: YAML
  TargetType: /AWS::EC2::Instance
  Name: replace_volume
  Content:
    schemaVersion: '0.3'
    description: 'Script for replacing an EBS data volume with a specified snapshot'
    parameters:
      VolumeId:
        type: String
        description: '(Required) Specify the VolumeId of the volume that needs to be replaced'
        allowedPattern: '^vol-[a-z0-9]{8,17}$'
      SnapshotId:
        type: String
        description: '(Required) Specify the SnapshotId of the snapshot to be used for the new
↪  volume'
        allowedPattern: '^snap-[a-z0-9]{8,17}$'
      DeleteVolume:
        type: Boolean
        description: '(Optional) Choose whether you want to delete the current volume'
        default: false
```

4.1. Get Volume Details

The first step in this automation is to retrieve the volume details using the `aws:executeAwsApi` action. The step calls the `DescribeVolumes` API with the `VolumeId` that is specified in the parameters section at the top of this document. When the volume details are retrieved, we can get the `InstanceId` and `DeviceName` that belongs to this volume. To ensure that the correct values are being used, I have chosen to get them from the volume instead of specifying them in the parameters section.

The `DeviceName` and `InstanceId` are retrieved using the `JSONPath` notation in the `API` `response` object and are pushed to the `outputs` section so it can be used in the next steps.

```yaml
# Retrieve EBS data volume details to get the DeviceName and InstanceId
- name: GetVolumeDetails
  action: aws:executeAwsApi
  timeoutSeconds: 120
  onFailure: Abort
  inputs:
    Service: ec2
    Api: DescribeVolumes
    VolumeIds:
      - '{{ VolumeId }}'
  outputs:
    - Name: DeviceName
      Selector: '$.Volumes[0].Attachments[0].Device'
    - Name: InstanceId
      Selector: '$.Volumes[0].Attachments[0].InstanceId'
  nextStep: GetInstanceDetails
```

After the volume details are successfully retrieved, the automation will continue to `GetInstanceDetails` step.

4.2. Get Instance Details

When the `DeviceName` and `InstanceId` are retrieved, we can fetch the instance details using the `DescribeInstances` API and the `InstanceId` from the previous step. To attach an `EBS` volume to an instance, it must reside within the same `AvailabilityZone` . In this step, I retrieve the `AvailabilityZone` and `hostname` using the `JSONPath` notation in the `API` `response` object. These values are pushed to the `outputs` section for further processing.

The `hostname` is required to give the correct `tags` (`Name` and `type`) to the new volume that's being created in the next step.

```yaml
# Retrieve instance details to get the AvailabilityZone and Hostname
- name: GetInstanceDetails
  action: aws:executeAwsApi
  timeoutSeconds: 120
  onFailure: Abort
  inputs:
    Service: ec2
    Api: DescribeInstances
    InstanceIds:
      - '{{ GetVolumeDetails.InstanceId }}'
  outputs:
    - Name: AvailabilityZone
      Selector: '$.Reservations[0].Instances[0].Placement.AvailabilityZone'
```

```
      Type: String
    - Name: Host
      Selector: $.Reservations[0].Instances[0].Tags[?(@.Key == 'Name')].Value
      Type: String
  nextStep: CreateVolume
```

When the instance details are succesfully retrieved, the script will move forward to the next step.

4.3. Create Volume

When the required details are retrieved, the automation will create an new `EBS` data volume based on the `snapshot` that's provided in the parameters section at the beginning of this document. To create an new volume, the automation uses the `CreateVolume` API with the `AvailabilityZone` and `SnapshotId` from the previous steps. This ensures that the volume is created within the same `AvailabilityZone` as the instance and the existing (snapshot) data is being used.

By default, the `EBS` data volume is created using the `gp2` volume type. This is an older drive with less performance. Therefore, I specified the `VolumeType` parameter and configured it to `gp3` to make sure the correct `VolumeType` is used. Lastly, the script adds the `Name` tag using the `hostname` that's retrieved in the previous step and the `data` keyword to easily identify that this volume is used as a data drive. The `Type` tag is added to make sure the volume is targeted by the `DLM` policy.

```
# Create new EBS data volume using the retrieved details and specified snapshot
- name: CreateVolume
  action: aws:executeAwsApi
  timeoutSeconds: 120
  onFailure: Abort
  inputs:
    Service: ec2
    Api: CreateVolume
    AvailabilityZone: '{{ GetInstanceDetails.AvailabilityZone }}'
    SnapshotId: '{{ SnapshotId }}'
    VolumeType: gp3
    TagSpecifications:
      - ResourceType: volume
        Tags:
         - Key: Name
           Value: '{{ GetInstanceDetails.Host }}-data'
         - Key: Type
           Value: '{{ GetInstanceDetails.Host }}-or-data-not-in-use'
   outputs:
    - Name: VolumeId
      Selector: '$.VolumeId'
      Type: String
   nextStep: WaitForVolumeCreation
```

When the volume is successfully created, the `VolumeId` will be pushed to the `outputs` section. It's retrieved using the `JSONPath` notation by targeting the `API` `response` object. Thereafter, the automation will continue to the `WaitForVolumeCreation` step.

4.4. Wait For Volume Creation

After the volume is successfully provisioned, the script needs to check if the volume is created successfully and available for attachment. In this step, I used the `aws:waitForAwsResourceProperty` action to wait for the volume's status becomes `available` To retrieve the status, I used the `DescribeVolumes` API with the `VolumeId` from the previous step. I then used the `PropertySelector` parameter to filter the `API` `response` object using the `JSONPath` notation.

```yaml
# Wait until the EBS data volume is succesfully created
- name: WaitForVolumeCreation
  action: aws:waitForAwsResourceProperty
  timeoutSeconds: 120
  onFailure: Abort
  inputs:
    Service: ec2
    Api: DescribeVolumes
    VolumeIds:
      - '{{ CreateVolume.VolumeId }}'
    PropertySelector: '$.Volumes[0].State'
    DesiredValues:
      - available
    nextStep: DetachVolume
```

When the volume's status is `available` , the value specified in the `DesiredValues` parameter, the volume is ready to be attached and the script will continue to the next step.

4.5. Detach Volume

When the automation reaches this step, the current `EBS` data volume will be detached from the instance. This part is handeled by an separate `SSM` automation document as some of the steps are using the `CommandType` command to execute `linux` commands on the `EC2` instance. The script uses the `aws:executeAutomation` action to start the `detach_volume` automation document. The parameters that are required by the document can be passed to the `RuntimeParameters` section. In this case, I specified the `VolumeId` from the parameters section at the top (current volume) to specify which volume needs to be detached.

```
# Detach the current EBS data volume
- name: DetachVolume
  action: aws:executeAutomation
  timeoutSeconds: 120
  onFailure: Abort
  isCritical: true
  inputs:
    DocumentName: detach_volume
    RuntimeParameters:
      VolumeId:
        - '{{ VolumeId }}'
   nextStep: WaitForVolumeDetachment
```

When the automation is successfully executed, the script continues to the next step to check for the execution status.

4.6. Wait for Volume Detachment

To make sure every step in the `detach_volume` automation is successfully executed. We need to check for the execution status before the new `EBS` data volume can be attached to the `EC2` instance. In this step `the aws:waitForAwsResourceProperty` is used again to check for the `SSM` execution status. I used the `GetAutomationExecution` API to retrieve the execution details with the `ExecutionId` that is available from the previous step.

```
# Wait until the current EBS data volume is succesfully detached
- name: WaitForVolumeDetachment
  action: aws:waitForAwsResourceProperty
  timeoutSeconds: 120
  onFailure: Abort
  inputs:
    Service: ssm
    Api: GetAutomationExecution
    AutomationExecutionId: '{{ DetachVolume.ExecutionId }}'
    PropertySelector: '$.AutomationExecution.AutomationExecutionStatus'
    DesiredValues:
      - Success
  nextStep: AttachVolume
```

With the `PropertySelector` I target the `AutomationExecutionStatus` using the `JSONPath` notation. When the document is successfully executed (`DesiredValues==Success`) the volume is successfully `umounted` and detached from the `EC2` instance. The automation will now continue to the next step, attaching the newly created `EBS` volume to the `EC2` instance.

4.7. Attach Volume

When the current `EBS` data volume is successfully detached from the `EC2` instance we can start attacing the newly created `EBS` data volume to the instance. This task is again handeled by an separate `SSM` automation document. The `attach_volume` document will be executed using the `aws:executeAutomation` action. The parameters: `DeviceName` , `VolumeId` and `InstanceId` are specified in the `RuntimeParameters` section and will be passed to the document.

The document is marked as critical using the `isCritical=true` parameter. When this step fails, the automation will fail and the other steps won't be executed.

```yaml
# Attach the newly created EBS data volume
- name: AttachVolume
  action: aws:executeAutomation
  timeoutSeconds: 120
  onFailure: Abort
  isCritical: true
  inputs:
    DocumentName: attach_volume
    RuntimeParameters:
      DeviceName:
        - '{{ GetVolumeDetails.DeviceName }}'
      VolumeId:
        - '{{ CreateVolume.VolumeId }}'
      InstanceId:
        - '{{ GetVolumeDetails.InstanceId }}'
  nextStep: WaitForVolumeAttachment
```

When the `attach_volume` document is successfully executed, the automation will continue to the next step to check for it's status.

4.8. Wait For Volume Attachment

After executing the `SSM` document we need to check it's status to make sure it's executed successfully. In this step, I use the `aws:waitForAwsResourceProperty` action again to check for the `Success` status. The execution details are retrieved using the `GetAutomationExecution` API with the `ExecutionId` from the previous step. With the `PropertySelector` I target the `AutomationExecutionStatus` using the `JSONPath` notation.

The script frequently polls the status and continues to the next step once the status has the value that's described in the `DesiredValues` section.

```yaml
# Wait until the newly created EBS data volume is succesfully attached
- name: WaitForVolumeAttachment
  action: aws:waitForAwsResourceProperty
  timeoutSeconds: 120
  onFailure: Abort
  inputs:
    Service: ssm
    Api: GetAutomationExecution
    AutomationExecutionId: '{{ AttachVolume.ExecutionId }}'
    PropertySelector: '$.AutomationExecution.AutomationExecutionStatus'
    DesiredValues:
      - Success
  nextStep: ChooseVolumeDeletion
```

When the status returns `Success` , the automation is executed successfully and the `EBS` data volume is attached to the instance. The automation will continue to the last step.

4.9. Choose Volume Deletion

Before this document is executed, you can choose whether to delete the current `EBS` data volume or keep it for later use. The automation will choose the next step based on the `DeleteVolume` parameter that is described at the top of this document. Using the `aws:branch` action, you can make a `switch/case` based on an specific value. In this case, when the `DeleteVolume` variable is equal to true (`BooleanEquals`) the step provided in the `NextStep` parameter will be executed, in this case the `DeleteVolume` step.

If the value is `false` , the script will end (`isEnd=true`) and the other steps won't be executed anymore, they keep the status `pending` .

```yaml
# Checks whether the old EBS data volume should be kept
- name: ChooseVolumeDeletion
  action: aws:branch
  inputs:
    Choices:
      - NextStep: DeleteVolume
        Variable: '{{ DeleteVolume }}'
        BooleanEquals: true
  isEnd: true
```

4.10. Delete Volume

When the `DeleteVolume` parameter is true, this step will be executed. The script uses the `DeleteVolume` API to delete the volume with the `VolumeId` specified in the parameters section at the top of this document (current volume) Unfortunately, this `API` call doesn't return anything. Therefore, we can't

check the status anymore. If no status is returned, the volume is deleted succesfully. When this step isn't executed successfully, the volume is not deleted.

```yaml
# Delete old EBS data volume if DeleteVolume variable is equal to true
- name: DeleteVolume
  action: aws:executeAwsApi
  timeoutSeconds: 120
  onFailure: Abort
  inputs:
    Service: ec2
    Api: DeleteVolume
    VolumeId: '{{ VolumeId }}'
```