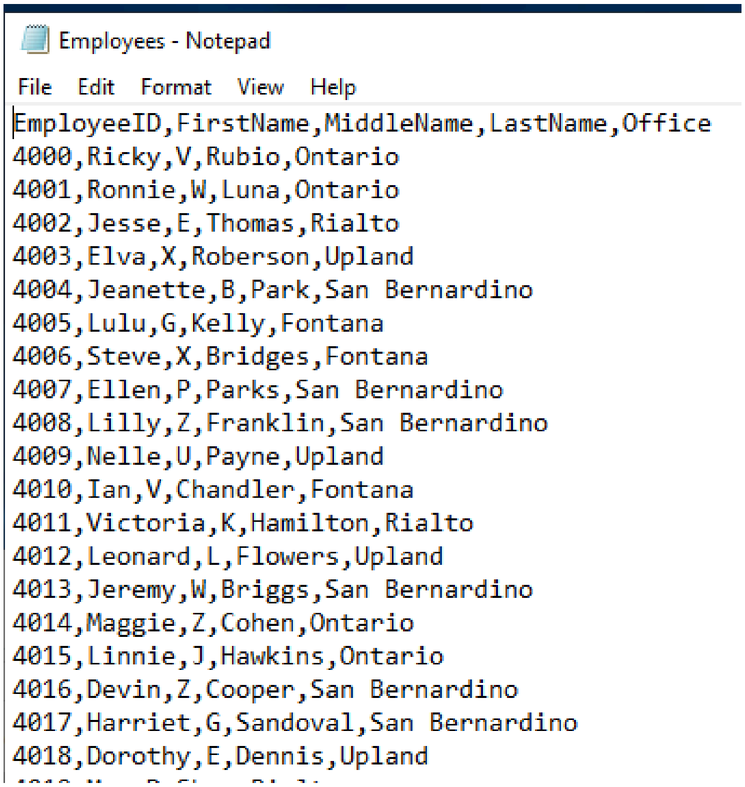
Ekene Onoh
April 15, 2024

# Automating Active Directory

To further my previous project regarding Active Directory Domains, I decided to dive into the realm of automating tasks using Powershell ISE. PowerShell scripts are a great way to streamline management and enhance efficiency. PowerShell's robust scripting capabilities can help you automate various administrative tasks, such as user provisioning, group management, and system monitoring, saving time and reducing the risk of errors. This project revolves around streamlining the user provisioning process within an Active Directory domain by leveraging PowerShell scripts to import employee information from a CSV file and automatically create corresponding user accounts. Leveraging PowerShell in a project will not only deepen one’s understanding of Active Directory but also develop valuable automation skills that can be applied to other IT projects.

## Importing the CSV

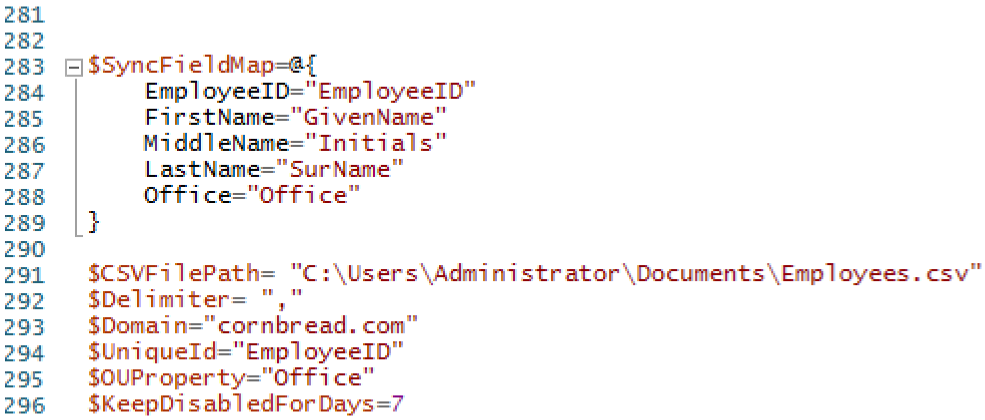


```
EmployeeID,FirstName,MiddleName,LastName,Office
4000,Ricky,V,Rubio,Ontario
4001,Ronnie,W,Luna,Ontario
4002,Jesse,E,Thomas,Rialto
4003,Elva,X,Roberson,Upland
4004,Jeanette,B,Park,San Bernardino
4005,Lulu,G,Kelly,Fontana
4006,Steve,X,Bridges,Fontana
4007,Ellen,P,Parks,San Bernardino
4008,Lilly,Z,Franklin,San Bernardino
4009,Nelle,U,Payne,Upland
4010,Ian,V,Chandler,Fontana
4011,Victoria,K,Hamilton,Rialto
4012,Leonard,L,Flowers,Upland
4013,Jeremy,W,Briggs,San Bernardino
4014,Maggie,Z,Cohen,Ontario
4015,Linnie,J,Hawkins,Ontario
4016,Devin,Z,Cooper,San Bernardino
4017,Harriet,G,Sandoval,San Bernardino
4018,Dorothy,E,Dennis,Upland
```

The first step of this project involves creating or obtaining a well-structured CSV file containing essential employee details. The file I used included ID numbers, first names, middle initials, last names, and office locations as the information for the employees. This CSV acts as the input data source for the PowerShell script. I utilized convertcsv.com to randomly generate the CSV file for authenticity purposes.

### The PowerShell Script

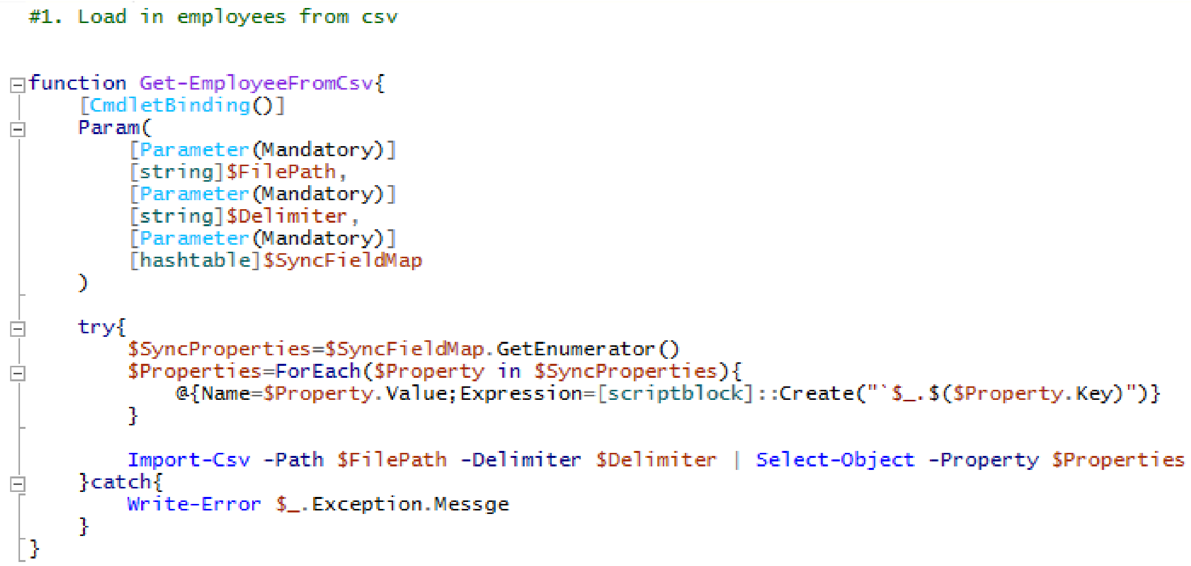
Next, a PowerShell script was developed to read the CSV file and iterate through each row, extracting the necessary information to create user accounts in Active Directory. The initial part of the code that was written was the SyncFieldMap along with other variables that were used throughout the script in various functions.



```


$SyncFieldMap=@{
    EmployeeID="EmployeeID"
    FirstName="GivenName"
    MiddleName="Initials"
    LastName="SurName"
    Office="Office"
}

$CSVFilePath= "C:\Users\Administrator\Documents\Employees.csv"
$Delimiter= ","
$Domain="cornbread.com"
$UniqueId="EmployeeID"
$OUProperty="Office"
$KeepDisabledForDays=7
```

The $SyncFieldMap variable is a hashtable that maps the column names in the CSV file to the corresponding Active Directory attributes. For example, the CSV column "EmployeeID" will be mapped to the Active Directory attribute "EmployeeID," "FirstName" to "GivenName," and so on. The $CSVFilePath variable specifies the path to the CSV file containing the employee information. This is the file from which the script will read the data. The $Delimiter variable specifies the delimiter used in the CSV file to separate the values. In this case, the delimiter is set to "," indicating that the CSV file is comma-separated. The $Domain variable specifies the domain name of the Active Directory where the user accounts will be created or updated. The $UniqueId variable specifies the unique identifier for each user in the CSV file. In this script, it is set to "EmployeeID," meaning that the "EmployeeID" column in the CSV file will be used to uniquely identify each user. The $OUProperty variable specifies the name of the column in the CSV file that contains the name of the organizational unit (OU) to which the user should be added. In this script, it is set to "Office," indicating that the "Office" column in the CSV file contains the OU names. The $KeepDisabledForDays variable specifies the number of days to keep a user account disabled before enabling it. This can be useful for temporarily disabling accounts without deleting them. Overall, this script provides a flexible and efficient way to synchronize user information from a CSV file to an Active Directory domain, using a customizable mapping of CSV columns to Active Directory attributes.

## Get-EmployeeFromCsv



```
#1. Load in employees from csv

function Get-EmployeeFromCsv{
    [CmdletBinding()]
    Param(
        [Parameter(Mandatory)]
        [string]$FilePath,
        [Parameter(Mandatory)]
        [string]$Delimiter,
        [Parameter(Mandatory)]
        [hashtable]$SyncFieldMap
    )

    try{
        $SyncProperties=$SyncFieldMap.GetEnumerator()
        $Properties=ForEach($Property in $SyncProperties){
            @{Name=$Property.Value;Expression=[scriptblock]::Create("`$_.$($Property.Key)")}
        }

        Import-Csv -Path $FilePath -Delimiter $Delimiter | Select-Object -Property $Properties
    }catch{
        Write-Error $_.Exception.Messge
    }
}
```

A function called “Get-EmployeeFromCsv” was defined to load employee information from a CSV file.

[CmdletBinding()]: This attribute indicates that the function supports common parameters, such as -Verbose, -Debug, -ErrorAction, etc.

Param(...): This block defines the parameters that the function expects to receive.

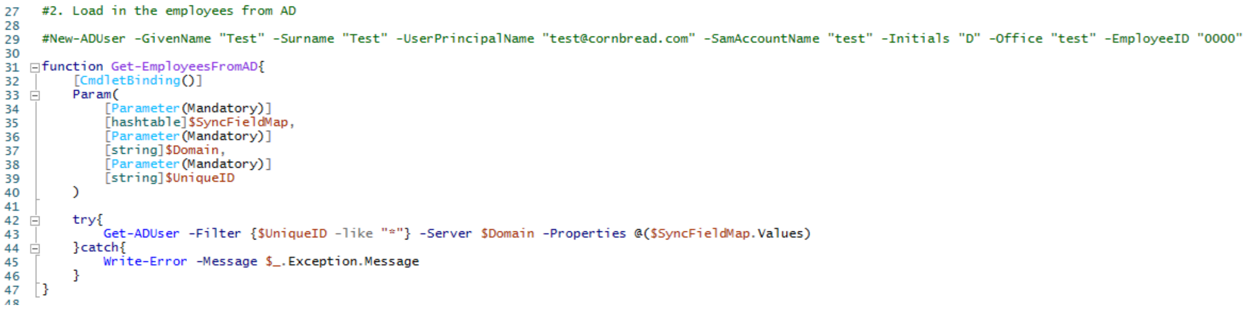
- $FilePath: The path to the CSV file containing the employee information.
- $Delimiter: The delimiter used in the CSV file to separate values.
- $SyncFieldMap: A hashtable that maps CSV column names to corresponding properties of the objects to be created.

The function starts by trying to execute the following code block:

- $SyncProperties=$SyncFieldMap.GetEnumerator(): This retrieves an enumerator for the hashtable, allowing the function to iterate over its key-value pairs.
- $Properties=ForEach($Property in $SyncProperties){...}: This loop iterates over each key-value pair in the hashtable and creates a new hashtable for each pair. Each new hashtable contains two properties: Name (the value from the hashtable) and Expression (a script block that accesses the corresponding CSV column value).

The script then uses the Import-Csv cmdlet to read the CSV file specified by $FilePath and uses the delimiter specified by $Delimiter. It pipes the output to Select-Object to create custom objects with properties specified by the $Properties hashtable. Each property is populated with the corresponding value from the CSV column based on the mapping defined in $SyncFieldMap. If any errors occur during the execution of the try block, the catch block catches the exception and writes the error message to the console using Write-Error.

## Get-EmployeeFromAD



```
#2. Load in the employees from AD

#New-ADUser -GivenName "Test" -Surname "Test" -UserPrincipalName "test@cornbread.com" -SamAccountName "test" -Initials "D" -Office "test" -EmployeeID "0000"

function Get-EmployeesFromAD{
    [CmdletBinding()]
    Param(
        [Parameter(Mandatory)]
        [hashtable]$SyncFieldMap,
        [Parameter(Mandatory)]
        [string]$Domain,
        [Parameter(Mandatory)]
        [string]$UniqueID
    )

    try{
        Get-ADUser -Filter {$UniqueID -like "*"} -Server $Domain -Properties @($SyncFieldMap.Values)
    }catch{
        Write-Error -Message $_.Exception.Message
    }
}
```

“Get-EmployeesFromAD” is a function that retrieves employee information from Active Directory based on specified criteria.

[CmdletBinding()]: This attribute indicates that the function supports common parameters, such as -Verbose, -Debug, -ErrorAction, etc.

Param(...): This block defines the parameters that the function expects to receive.

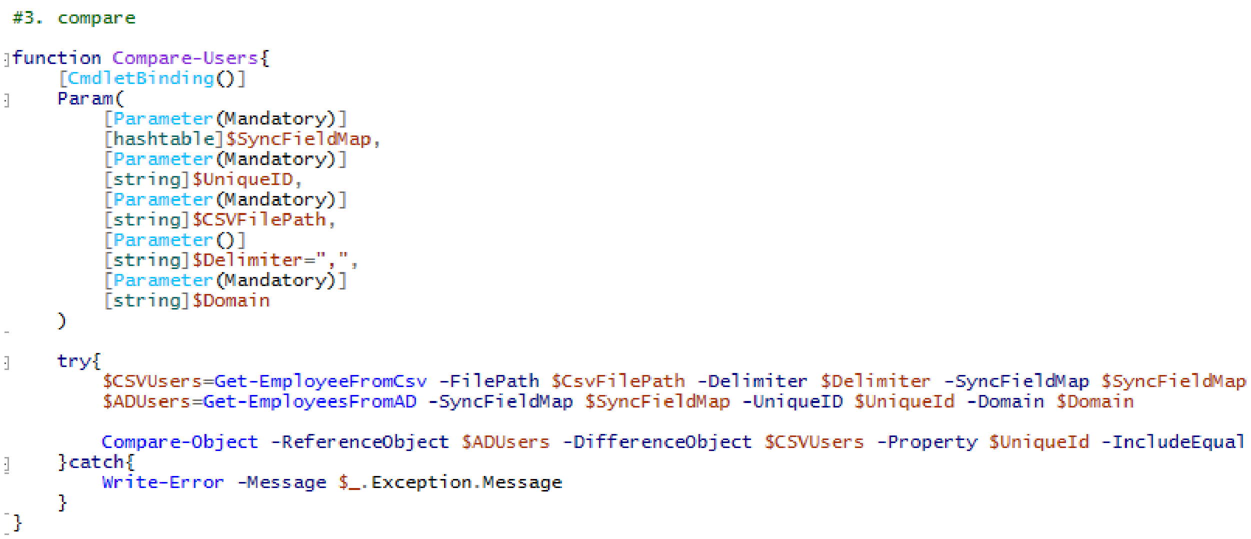
- $SyncFieldMap: A hashtable that maps properties of the objects to be retrieved from Active Directory to their corresponding names.
- $Domain: The domain name of the Active Directory from which to retrieve the information.
- $UniqueID: The unique identifier used to filter the search in Active Directory.

The function starts by trying to execute the following code block:

- “Get-ADUser -Filter {$UniqueID -like "*"} -Server $Domain -Properties @($SyncFieldMap.Values)”: This cmdlet retrieves user objects from Active Directory based on the specified filter ($UniqueID -like "*", which essentially retrieves all users) and server ($Domain). The -Properties parameter is used to specify which properties of the user objects to retrieve, based on the values in the $SyncFieldMap hashtable.

If any errors occur during the execution of the try block, the catch block catches the exception and writes the error message to the console using Write-Error.

Compare-Users



```
#3. compare

function Compare-Users{
    [CmdletBinding()]
    Param(
        [Parameter(Mandatory)]
        [hashtable]$SyncFieldMap,
        [Parameter(Mandatory)]
        [string]$UniqueID,
        [Parameter(Mandatory)]
        [string]$CSVFilePath,
        [Parameter()]
        [string]$Delimiter=",",
        [Parameter(Mandatory)]
        [string]$Domain
    )

    try{
        $CSVUsers=Get-EmployeeFromCsv -FilePath $CsvFilePath -Delimiter $Delimiter -SyncFieldMap $SyncFieldMap
        $ADUsers=Get-EmployeesFromAD -SyncFieldMap $SyncFieldMap -UniqueID $UniqueId -Domain $Domain

        Compare-Object -ReferenceObject $ADUsers -DifferenceObject $CSVUsers -Property $UniqueId -IncludeEqual
    }catch{
        Write-Error -Message $_.Exception.Message
    }
}
```

The function “Compare-Users” compares and distinguishes which employees exist in either the CSV file, the Active Directory domain, or both.

[CmdletBinding()]: This attribute indicates that the function supports common parameters, such as -Verbose, -Debug, -ErrorAction, etc.

Param(...): This block defines the parameters that the function expects to receive.

- $SyncFieldMap: A hashtable that maps properties of the objects to be compared between the CSV file and Active Directory.
- $UniqueID: The unique identifier used to identify employees in both the CSV file and Active Directory.
- $CSVFilePath: The path to the CSV file containing the employee information.
- $Delimiter: The delimiter used in the CSV file to separate values (default is ",").
- $Domain: The domain name of the Active Directory.

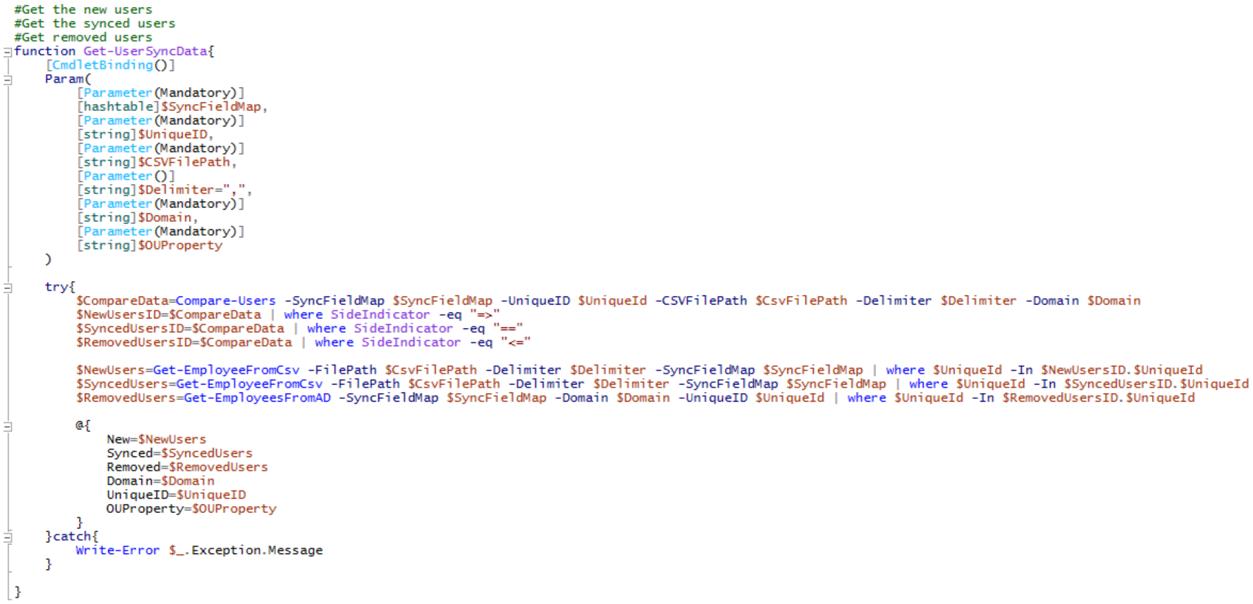
The function starts by trying to execute the following code block:

- “$CSVUsers=Get-EmployeeFromCsv -FilePath $CsvFilePath -Delimiter $Delimiter -SyncFieldMap $SyncFieldMap”: This line calls the Get-EmployeeFromCsv function (which you previously described) to retrieve employee information from the CSV file.
- “$ADUsers=Get-EmployeesFromAD -SyncFieldMap $SyncFieldMap -UniqueID $UniqueId -Domain $Domain”: This line calls the Get-EmployeesFromAD function (which you also described) to retrieve employee information from Active Directory.

The Compare-Object cmdlet is then used to compare the employee objects from the CSV file ($CSVUsers) and Active Directory ($ADUsers) based on the unique identifier ($UniqueId). The -IncludeEqual parameter is used to include objects that are equal in both sets (i.e., employees that exist in both the CSV file and Active Directory) in the output.

If any errors occur during the execution of the try block, the catch block catches the exception and writes the error message to the console using Write-Error.

## Get-UserSyncData



```
#Get the new users
#Get the synced users
#Get removed users
function Get-UserSyncData{
    [CmdletBinding()]
    Param(
        [Parameter(Mandatory)]
        [hashtable]$SyncFieldMap,
        [Parameter(Mandatory)]
        [string]$UniqueID,
        [Parameter(Mandatory)]
        [string]$CSVFilePath,
        [Parameter()]
        [string]$Delimiter=",",
        [Parameter(Mandatory)]
        [string]$Domain,
        [Parameter(Mandatory)]
        [string]$OUProperty
    )

    try{
        $CompareData=Compare-Users -SyncFieldMap $SyncFieldMap -UniqueID $UniqueId -CSVFilePath $CsvFilePath -Delimiter $Delimiter -Domain $Domain
        $NewUsersID=$CompareData | where SideIndicator -eq "=>"
        $SyncedUsersID=$CompareData | where SideIndicator -eq "=="
        $RemovedUsersID=$CompareData | where SideIndicator -eq "<="

        $NewUsers=Get-EmployeeFromCsv -FilePath $CsvFilePath -Delimiter $Delimiter -SyncFieldMap $SyncFieldMap | where $UniqueId -In $NewUsersID.$UniqueId
        $SyncedUsers=Get-EmployeeFromCsv -FilePath $CsvFilePath -Delimiter $Delimiter -SyncFieldMap $SyncFieldMap | where $UniqueId -In $SyncedUsersID.$UniqueId
        $RemovedUsers=Get-EmployeesFromAD -SyncFieldMap $SyncFieldMap -Domain $Domain -UniqueID $UniqueId | where $UniqueId -In $RemovedUsersID.$UniqueId

        @{
            New=$NewUsers
            Synced=$SyncedUsers
            Removed=$RemovedUsers
            Domain=$Domain
            UniqueID=$UniqueID
            OUProperty=$OUProperty
        }
    }catch{
        Write-Error $_.Exception.Message
    }

}
```

This PowerShell script function called Get-UserSyncData synchronizes employee data between a CSV file and an Active Directory domain. It checks which employees need to be added to the Active Directory domain and which users have been removed.

[CmdletBinding()]: This attribute indicates that the function supports common parameters, such as -Verbose, -Debug, -ErrorAction, etc.

Param(...): This block defines the parameters that the function expects to receive.

- $SyncFieldMap: A hashtable that maps properties of the objects to be synchronized between the CSV file and Active Directory.
- $UniqueID: The unique identifier used to identify employees in both the CSV file and Active Directory.

- $CSVFilePath: The path to the CSV file containing the employee information.
- $Delimiter: The delimiter used in the CSV file to separate values (default is ",").
- $Domain: The domain name of the Active Directory.
- $OUProperty: The name of the property in the CSV file that contains the name of the organizational unit (OU) to which the user should be added.

The function starts by trying to execute the following code block:

- "$CompareData=Compare-Users -SyncFieldMap $SyncFieldMap -UniqueID $UniqueId -CSVFilePath $CsvFilePath -Delimiter $Delimiter -Domain $Domain": This line calls the Compare-Users function (which you previously described) to compare employee data between the CSV file and Active Directory.
- The script then filters the $CompareData variable to separate employees that need to be added ($NewUsersID), employees that are already synced ($SyncedUsersID), and employees that have been removed ($RemovedUsersID).

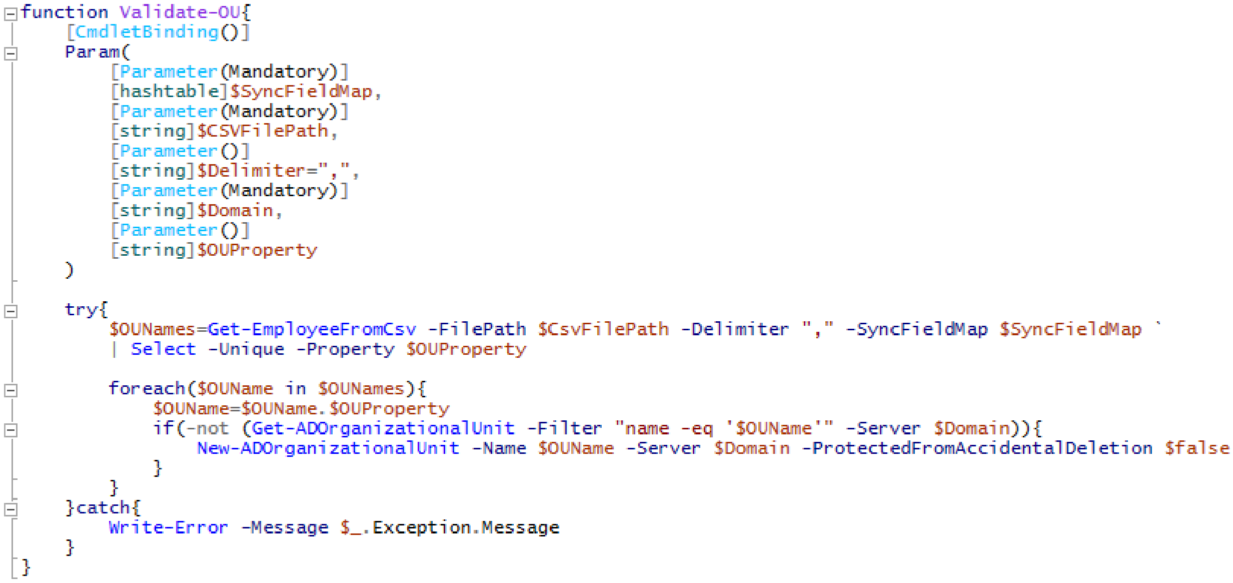
The script then retrieves the details of new, synced, and removed users based on the filtered data:

- $NewUsers: Retrieves new users from the CSV file that need to be added to Active Directory.
- $SyncedUsers: Retrieves users from the CSV file that are already synced with Active Directory.
- $RemovedUsers: Retrieves users from Active Directory that have been removed from the CSV file.

Finally, the function returns a hashtable containing the new, synced, and removed users, along with the domain, unique identifier, and OU property.

If any errors occur during the execution of the try block, the catch block catches the exception and writes the error message to the console using Write-Error.

Validate-OU



```
function Validate-OU{
    [CmdletBinding()]
    Param(
        [Parameter(Mandatory)]
        [hashtable]$SyncFieldMap,
        [Parameter(Mandatory)]
        [string]$CSVFilePath,
        [Parameter()]
        [string]$Delimiter=",",
        [Parameter(Mandatory)]
        [string]$Domain,
        [Parameter()]
        [string]$OUProperty
    )

    try{
        $OUNames=Get-EmployeeFromCsv -FilePath $CsvFilePath -Delimiter "," -SyncFieldMap $SyncFieldMap `
        | Select -Unique -Property $OUProperty

        foreach($OUName in $OUNames){
            $OUName=$OUName.$OUProperty
            if(-not (Get-ADOrganizationalUnit -Filter "name -eq '$OUName'" -Server $Domain)){
                New-ADOrganizationalUnit -Name $OUName -Server $Domain -ProtectedFromAccidentalDeletion $false
            }
        }
    }catch{
        Write-Error -Message $_.Exception.Message
    }
}
```

“Validate-OU” checks and ensures that employee users within the Active Directory are placed and added to the correct organizational unit (OU).

[CmdletBinding()]: This attribute indicates that the function supports common parameters, such as -Verbose, -Debug, -ErrorAction, etc.

Param(...): This block defines the parameters that the function expects to receive.

- $SyncFieldMap: A hashtable that maps properties of the objects to be synchronized between the CSV file and Active Directory.
- $CSVFilePath: The path to the CSV file containing the employee information.
- $Delimiter: The delimiter used in the CSV file to separate values (default is ",").
- $Domain: The domain name of the Active Directory.
- $OUProperty: The name of the property in the CSV file that contains the name of the organizational unit (OU) to which the user should be added.

The function starts by trying to execute the following code block:

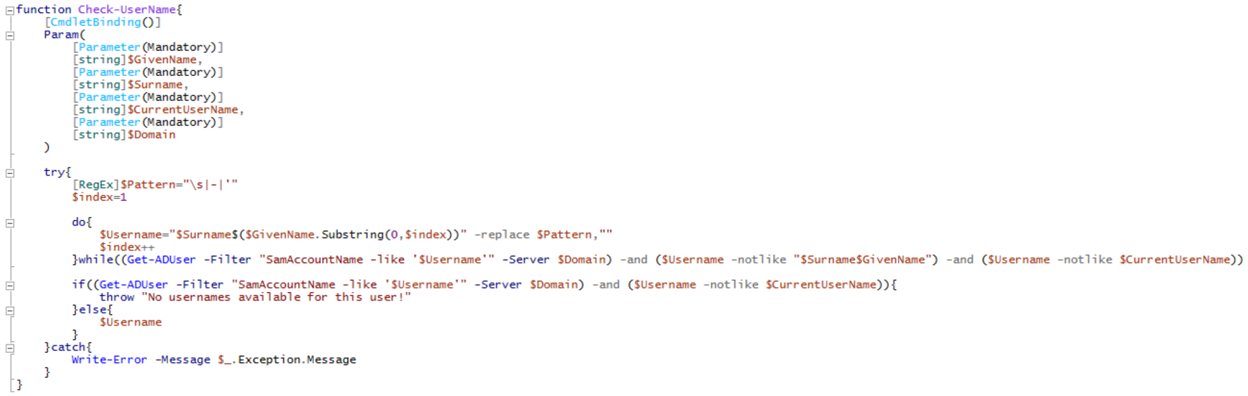
- “$OUNames=Get-EmployeeFromCsv -FilePath $CsvFilePath -Delimiter "," -SyncFieldMap $SyncFieldMap | Select -Unique -Property $OUProperty”: This line calls the Get-EmployeeFromCsv function (which you previously described) to retrieve unique organizational unit names from the CSV file.

The script then iterates over each unique OU name and checks if the OU already exists in Active Directory. If the OU does not exist, it creates a new OU using the New-ADOrganizationalUnit cmdlet.

- “if(-not (Get-ADOrganizationalUnit -Filter "name -eq '$OUName'" -Server $Domain))”: This line checks if the OU with the name stored in $OUName does not exist.
- “New-ADOrganizationalUnit -Name $OUName -Server $Domain -ProtectedFromAccidentalDeletion $false”: This line creates a new OU with the name stored in $OUName, on the specified $Domain, and disables protection from accidental deletion.

If any errors occur during the execution of the try block, the catch block catches the exception and writes the error message to the console using Write-Error.

Check-Username



```
function Check-UserName{
    [CmdletBinding()]
    Param(
        [Parameter(Mandatory)]
        [string]$GivenName,
        [Parameter(Mandatory)]
        [string]$Surname,
        [Parameter(Mandatory)]
        [string]$CurrentUserName,
        [Parameter(Mandatory)]
        [string]$Domain
    )

    try{
        [RegEx]$Pattern="\s|-|'"
        $index=1

        do{
            $Username="$Surname$($GivenName.Substring(0,$index))" -replace $Pattern,""
            $index++
        }while((Get-ADUser -Filter "SamAccountName -like '$Username'" -Server $Domain) -and ($Username -notlike "$Surname$GivenName") -and ($Username -notlike $CurrentUserName))

        if((Get-ADUser -Filter "SamAccountName -like '$Username'" -Server $Domain) -and ($Username -notlike $CurrentUserName)){
            throw "No usernames available for this user!"
        }else{
            $Username
        }
    }catch{
        Write-Error -Message $_.Exception.Message
    }
}
```

“Check-UserName” checks for the availability of usernames for employees being added to the Active Directory domain. The usernames are created using a specific format: the employee's last name followed by their first initial.

[CmdletBinding()]: This attribute indicates that the function supports common parameters, such as -Verbose, -Debug, -ErrorAction, etc.

Param(...): This block defines the parameters that the function expects to receive.

- $GivenName: The first name of the employee.
- $Surname: The last name of the employee.
- $CurrentUserName: The current username of the employee (if applicable).
- $Domain: The domain name of the Active Directory.

The function starts by trying to execute the following code block:

- [RegEx]$Pattern="\s|-|'": This line defines a regular expression pattern that matches whitespace, hyphens, and single quotes.
- $index=1: This initializes a variable $index to 1, which will be used to iterate through the first initial of the employee's first name.

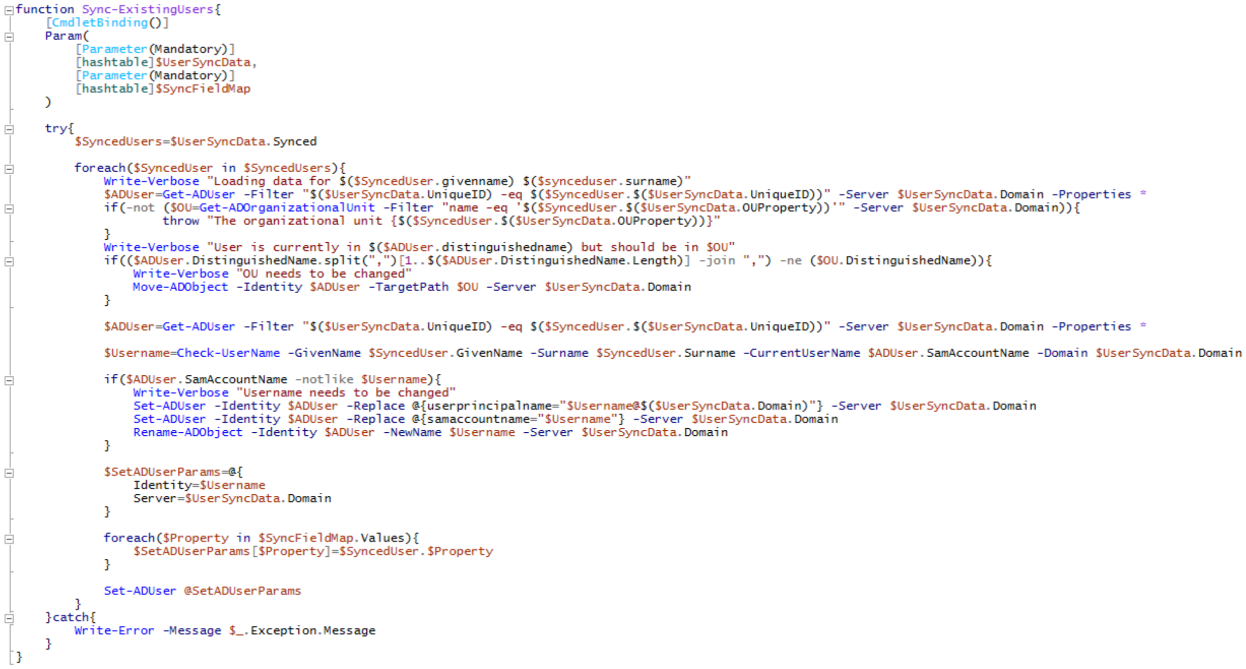
The script then enters a do loop that iterates until it finds an available username:

- Inside the loop, it constructs a potential username using the employee's last name and the first initial of their first name, removing any characters that match the $Pattern regex.
- It then increments $index to move to the next character in the first name.
- The loop continues until it finds a username that meets the following conditions:
    - The username is not already in use ((Get-ADUser -Filter "SamAccountName -like '$Username'" -Server $Domain)).
    - The username is not the same as the current username (($Username -notlike $CurrentUserName)).

If no available username is found, the function throws an error message stating that no usernames are available for the user. Otherwise, it returns the available username.

If any errors occur during the execution of the try block, the catch block catches the exception and writes the error message to the console using Write-Error.

## Sync-ExistingUsers



```
function Sync-ExistingUsers{
    [CmdletBinding()]
    Param(
        [Parameter(Mandatory)]
        [hashtable]$UserSyncData,
        [Parameter(Mandatory)]
        [hashtable]$SyncFieldMap
    )

    try{
        $SyncedUsers=$UserSyncData.Synced

        foreach($SyncedUser in $SyncedUsers){
            Write-Verbose "Loading data for $($SyncedUser.givenname) $($synceduser.surname)"
            $ADUser=Get-ADUser -Filter "$($UserSyncData.UniqueID) -eq $($SyncedUser.$($UserSyncData.UniqueID))" -Server $UserSyncData.Domain -Properties *
            if(-not ($OU=Get-ADOrganizationalUnit -Filter "name -eq '$($SyncedUser.$($UserSyncData.OUProperty))'" -Server $UserSyncData.Domain)){
                    throw "The organizational unit {$($SyncedUser.$($UserSyncData.OUProperty))}"
            }
            Write-Verbose "User is currently in $($ADUser.distinguishedname) but should be in $OU"
            if(($ADUser.DistinguishedName.split(",")[1..$($ADUser.DistinguishedName.Length)] -join ",") -ne ($OU.DistinguishedName)){
                Write-Verbose "OU needs to be changed"
                Move-ADObject -Identity $ADUser -TargetPath $OU -Server $UserSyncData.Domain
            }

            $ADUser=Get-ADUser -Filter "$($UserSyncData.UniqueID) -eq $($SyncedUser.$($UserSyncData.UniqueID))" -Server $UserSyncData.Domain -Properties *

            $Username=Check-UserName -GivenName $SyncedUser.GivenName -Surname $SyncedUser.Surname -CurrentUserName $ADUser.SamAccountName -Domain $UserSyncData.Domain

            if($ADUser.SamAccountName -notlike $Username){
                Write-Verbose "Username needs to be changed"
                Set-ADUser -Identity $ADUser -Replace @{userprincipalname="$Username@$($UserSyncData.Domain)"} -Server $UserSyncData.Domain
                Set-ADUser -Identity $ADUser -Replace @{samaccountname="$Username"} -Server $UserSyncData.Domain
                Rename-ADObject -Identity $ADUser -NewName $Username -Server $UserSyncData.Domain
            }

            $SetADUserParams=@{
                Identity=$Username
                Server=$UserSyncData.Domain
            }

            foreach($Property in $SyncFieldMap.Values){
                $SetADUserParams[$Property]=$SyncedUser.$Property
            }

            Set-ADUser @SetADUserParams
        }
    }catch{
        Write-Error -Message $_.Exception.Message
    }
}
```

"Sync-ExistingUsers" is used to check that the employee users in the Active Directory are synced properly with the current version of the employee CSV file.

[CmdletBinding()]: This attribute indicates that the function supports common parameters, such as -Verbose, -Debug, -ErrorAction, etc.

Param(...): This block defines the parameters that the function expects to receive.

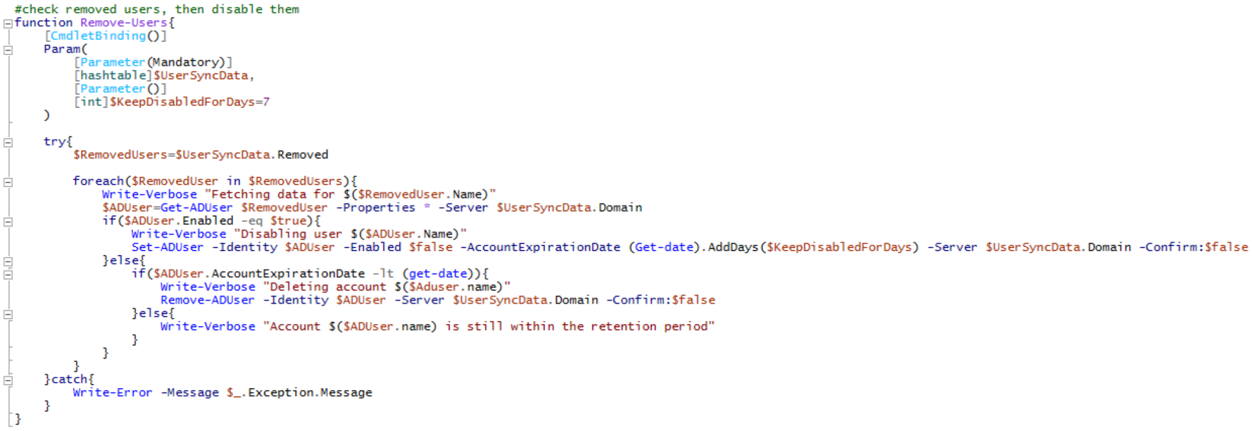
- $UserSyncData: A hashtable containing synchronization data for users, including information such as which users are synced (Synced), which users are new (New), which users are removed (Removed), the domain name (Domain), the unique identifier for users (UniqueID), and the organizational unit property name (OUProperty).
- $SyncFieldMap: A hashtable that maps properties of the objects to be synchronized between the CSV file and Active Directory.

The function starts by trying to execute the following code block:

- It retrieves the synced users from the $UserSyncData.Synced hashtable.
- It iterates over each synced user and performs the following operations:
    - Loads the Active Directory user object for the synced user using the unique identifier and domain name.

- ○ Checks if the organizational unit (OU) for the user exists in Active Directory. If it doesn't exist, the function throws an error.
- ○ Checks if the user's current OU matches the expected OU based on the CSV file. If it doesn't match, it moves the user to the correct OU.
- ○ Checks if the user's username (SamAccountName) matches the expected username based on the CSV file. If it doesn't match, it changes the username.
- ○ Updates the user's properties based on the mapping defined in $SyncFieldMap.
- ○ If any errors occur during these operations, the function catches the exception and writes the error message to the console using Write-Error.

## Remove-Users



```
#check removed users, then disable them
function Remove-Users{
    [CmdletBinding()]
    Param(
        [Parameter(Mandatory)]
        [hashtable]$UserSyncData,
        [Parameter()]
        [int]$KeepDisabledForDays=7
    )

    try{
        $RemovedUsers=$UserSyncData.Removed

        foreach($RemovedUser in $RemovedUsers){
            Write-Verbose "Fetching data for $($RemovedUser.Name)"
            $ADUser=Get-ADUser $RemovedUser -Properties * -Server $UserSyncData.Domain
            if($ADUser.Enabled -eq $true){
                Write-Verbose "Disabling user $($ADUser.Name)"
                Set-ADUser -Identity $ADUser -Enabled $false -AccountExpirationDate (Get-date).AddDays($KeepDisabledForDays) -Server $UserSyncData.Domain -Confirm:$false
            }else{
                if($ADUser.AccountExpirationDate -lt (get-date)){
                    Write-Verbose "Deleting account $($Aduser.name)"
                    Remove-ADUser -Identity $ADUser -Server $UserSyncData.Domain -Confirm:$false
                }else{
                    Write-Verbose "Account $($ADUser.name) is still within the retention period"
                }
            }
        }
    }catch{
        Write-Error -Message $_.Exception.Message
    }
}
```

The function “Remove-Users “checks which users have been removed from the CSV file, disables them and eventually deletes them from the Active Directory Domain when their retention period is over.

[CmdletBinding()]: This attribute indicates that the function supports common parameters, such as -Verbose, -Debug, -ErrorAction, etc.

Param(...): This block defines the parameters that the function expects to receive.

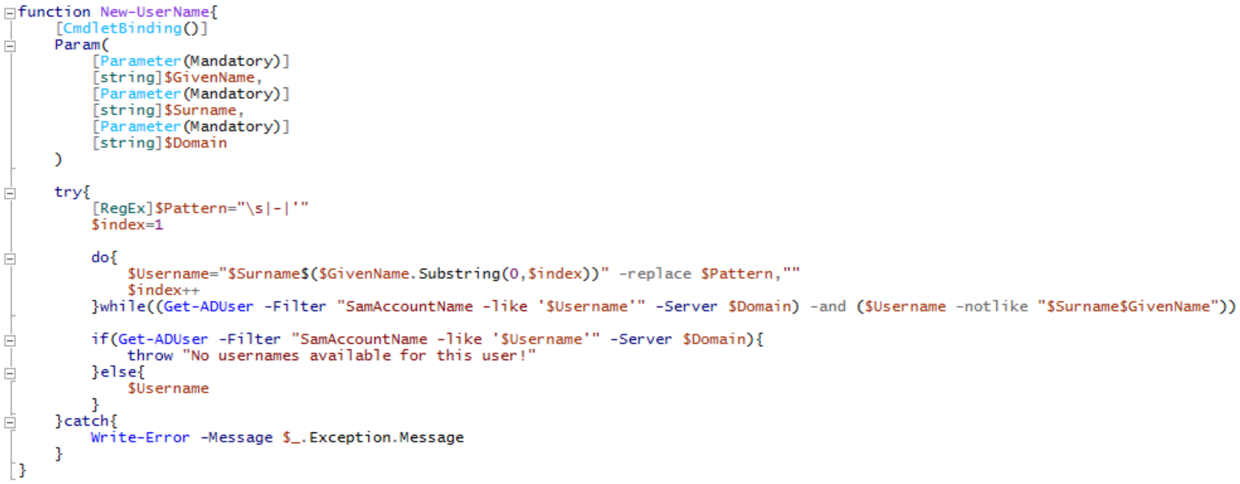
- $UserSyncData: A hashtable containing synchronization data for users, including information such as which users are removed (Removed) and the domain name (Domain).
- $KeepDisabledForDays: An optional parameter that specifies the number of days to keep disabled user accounts before deleting them (default is 7 days).

The function starts by trying to execute the following code block:

- It retrieves the removed users from the $UserSyncData.Removed hashtable.
- It iterates over each removed user and performs the following operations:
    - Fetches the Active Directory user object for the removed user.
    - Checks if the user is currently enabled. If the user is enabled, it disables the user account and sets an account expiration date based on the retention period.
    - If the user is already disabled, it checks if the account expiration date is in the past. If it is, it deletes the user account from Active Directory. If the account expiration date is still in the future, it logs a message indicating that the account is still within the retention period.

If any errors occur during the execution of the try block, the catch block catches the exception and writes the error message to the console using Write-Error.

## New-UserName



```
function New-UserName{
    [CmdletBinding()]
    Param(
        [Parameter(Mandatory)]
        [string]$GivenName,
        [Parameter(Mandatory)]
        [string]$Surname,
        [Parameter(Mandatory)]
        [string]$Domain
    )

    try{
        [RegEx]$Pattern="\s|-|'"
        $index=1

        do{
            $Username="$Surname$($GivenName.Substring(0,$index))" -replace $Pattern,""
            $index++
        }while((Get-ADUser -Filter "SamAccountName -like '$Username'" -Server $Domain) -and ($Username -notlike "$Surname$GivenName"))

        if(Get-ADUser -Filter "SamAccountName -like '$Username'" -Server $Domain){
            throw "No usernames available for this user!"
        }else{
            $Username
        }
    }catch{
        Write-Error -Message $_.Exception.Message
    }
}
```

“New-UserName” creates new usernames for employees using a specific format: the employee's last name followed by their first initial.

[CmdletBinding()]: This attribute indicates that the function supports common parameters, such as -Verbose, -Debug, -ErrorAction, etc.

Param(...): This block defines the parameters that the function expects to receive.

- $GivenName: The first name of the employee.
- $Surname: The last name of the employee.
- $Domain: The domain name of the Active Directory.

The function starts by trying to execute the following code block:

- [RegEx]$Pattern="\s|-|'": This line defines a regular expression pattern that matches whitespace, hyphens, and single quotes.
- $index=1: This initializes a variable $index to 1, which will be used to iterate through the first initial of the employee's first name.

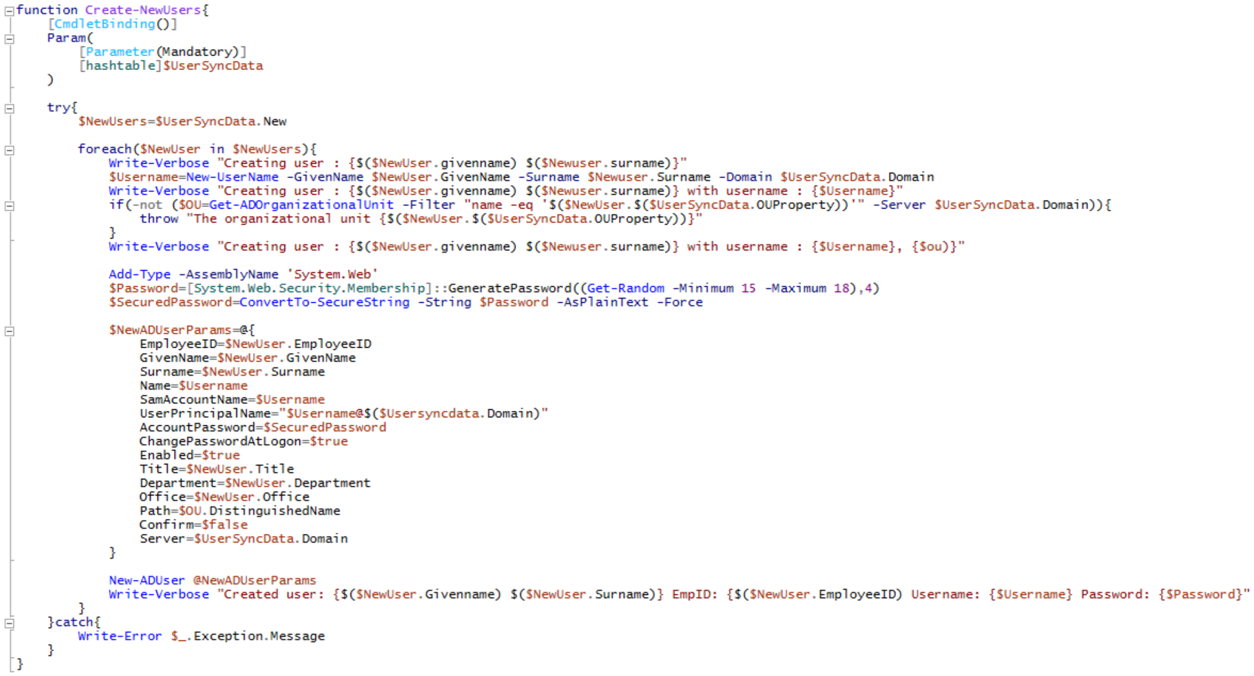
The script then enters a do loop that iterates until it finds an available username:

- Inside the loop, it constructs a potential username using the employee's last name and the first initial of their first name, removing any characters that match the $Pattern regex.
- It then increments $index to move to the next character in the first name.
- The loop continues until it finds a username that meets the following conditions:
    - The username is not already in use ((Get-ADUser -Filter "SamAccountName -like '$Username'" -Server $Domain)).
    - The username is not the same as the combination of the last name and first name (($Username -notlike "$Surname$GivenName")).

If no available username is found, the function throws an error message stating that no usernames are available for the user. Otherwise, it returns the available username.

If any errors occur during the execution of the try block, the catch block catches the exception and writes the error message to the console using Write-Error.

## Create-NewUsers



```
function Create-NewUsers{
    [CmdletBinding()]
    Param(
        [Parameter(Mandatory)]
        [hashtable]$UserSyncData
    )

    try{
        $NewUsers=$UserSyncData.New

        foreach($NewUser in $NewUsers){
            Write-Verbose "Creating user : {$($NewUser.givenname) $($Newuser.surname)}"
            $Username=New-UserName -GivenName $NewUser.GivenName -Surname $Newuser.Surname -Domain $UserSyncData.Domain
            Write-Verbose "Creating user : {$($NewUser.givenname) $($Newuser.surname)} with username : {$Username}"
            if(-not ($OU=Get-ADOrganizationalUnit -Filter "name -eq '$($NewUser.$($UserSyncData.OUProperty))'" -Server $UserSyncData.Domain)){
                throw "The organizational unit {$($NewUser.$($UserSyncData.OUProperty))}"
            }
            Write-Verbose "Creating user : {$($NewUser.givenname) $($Newuser.surname)} with username : {$Username}, {$ou)}"

            Add-Type -AssemblyName 'System.Web'
            $Password=[System.Web.Security.Membership]::GeneratePassword((Get-Random -Minimum 15 -Maximum 18),4)
            $SecuredPassword=ConvertTo-SecureString -String $Password -AsPlainText -Force

            $NewADUserParams=@{
                EmployeeID=$NewUser.EmployeeID
                GivenName=$NewUser.GivenName
                Surname=$NewUser.Surname
                Name=$Username
                SamAccountName=$Username
                UserPrincipalName="$Username@$($Usersyncdata.Domain)"
                AccountPassword=$SecuredPassword
                ChangePasswordAtLogon=$true
                Enabled=$true
                Title=$NewUser.Title
                Department=$NewUser.Department
                Office=$NewUser.Office
                Path=$OU.DistinguishedName
                Confirm=$false
                Server=$UserSyncData.Domain
            }

            New-ADUser @NewADUserParams
            Write-Verbose "Created user: {$($NewUser.Givenname) $($NewUser.Surname)} EmpID: {$($NewUser.EmployeeID) Username: {$Username} Password: {$Password}"
        }
    }catch{
        Write-Error $_.Exception.Message
    }
}
```

"Create-NewUsers" is used to create new employee user accounts within the Active Directory Domain.

[CmdletBinding()]: This attribute indicates that the function supports common parameters, such as -Verbose, -Debug, -ErrorAction, etc.

Param(...): This block defines the parameters that the function expects to receive.

- $UserSyncData: A hashtable containing synchronization data for users, including information such as which users are new (New), the domain name (Domain), and the organizational unit property name (OUProperty).

The function starts by trying to execute the following code block:

- It retrieves the new users from the $UserSyncData.New hashtable.
- It iterates over each new user and performs the following operations:
    - Creates a new username for the user using the New-UserName function, which you previously described, based on the user's first name and last name.
    - Retrieves the organizational unit (OU) for the user. If the OU does not exist, the function throws an error.
    - Generates a random password for the user using the GeneratePassword method from the System.Web.Security.Membership class.

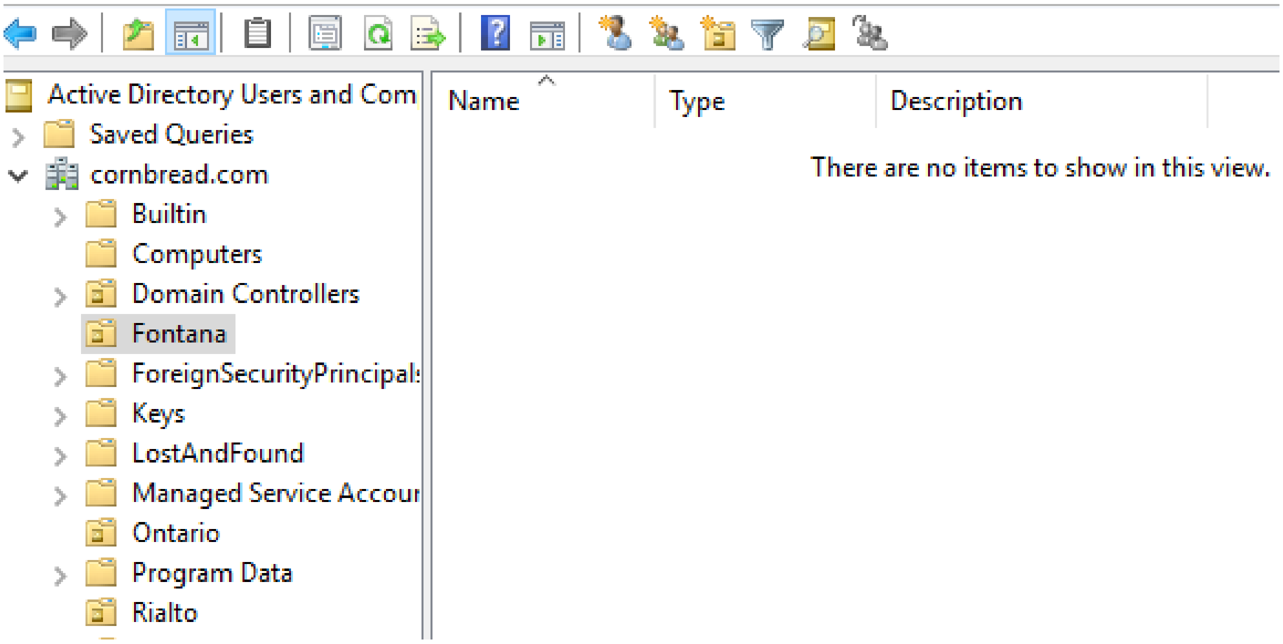
    - Creates a secure string from the generated password.
    - Defines parameters for creating a new user account in Active Directory, including the employee ID, given name, surname, username, user principal name, password, and other properties.
    - Creates a new user account in Active Directory using the New-ADUser cmdlet with the defined parameters.

If any errors occur during the execution of the try block, the catch block catches the exception and writes the error message to the console using Write-Error.

**Testing the Script**

To test the full script, I ran it in PowerShell ISE where I had my created Active Directory Domain imported as a variable. I ensured that the CSV file containing employee information was correctly formatted and had the necessary data. I then executed the script, which created new employee user accounts in the Active Directory Domain based on the information in the CSV file. After running the script, I verified in the Active Directory Users and Computers console that the new user accounts were created and placed in the correct organizational units. The successful creation of the new user accounts indicated that the script worked as intended and effectively synchronized the employee data with the Active Directory Domain.

Active Directory Before

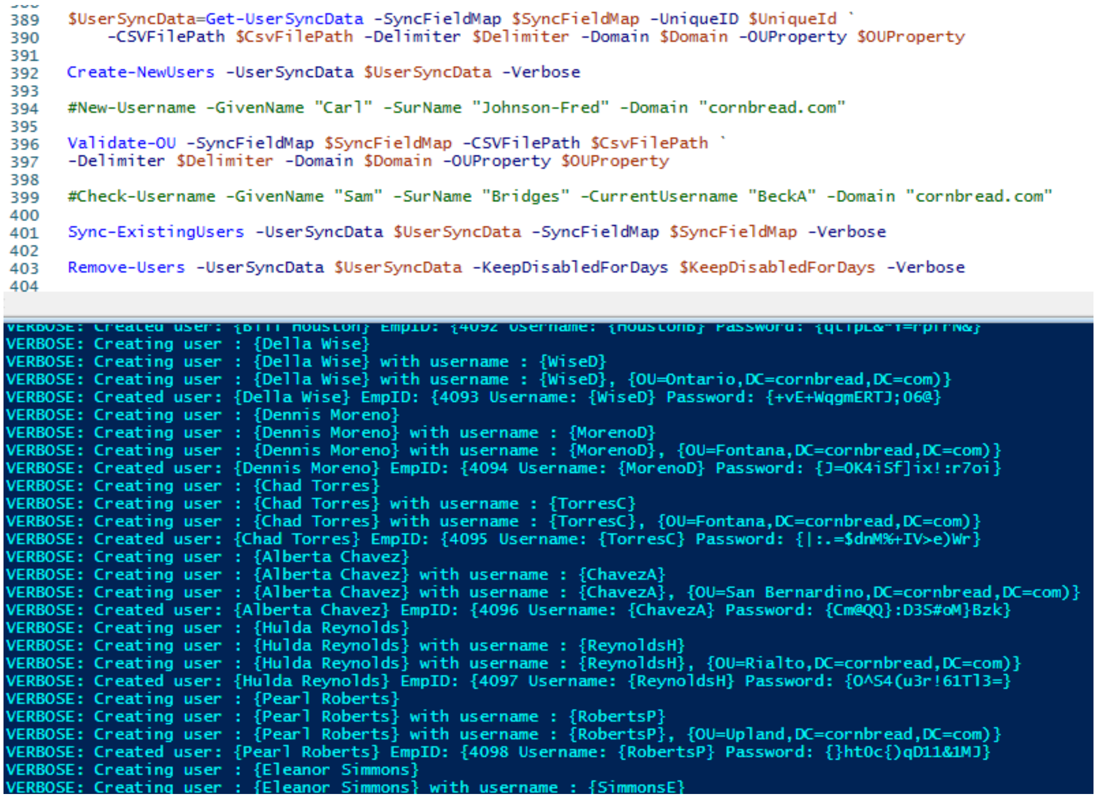


Active Directory Users and Com
- Saved Queries
- cornbread.com
  - Builtin
  - Computers
  - Domain Controllers
  - Fontana
  - ForeignSecurityPrincipal:
  - Keys
  - LostAndFound
  - Managed Service Accour
  - Ontario
  - Program Data
  - Rialto

Name | Type | Description

There are no items to show in this view.

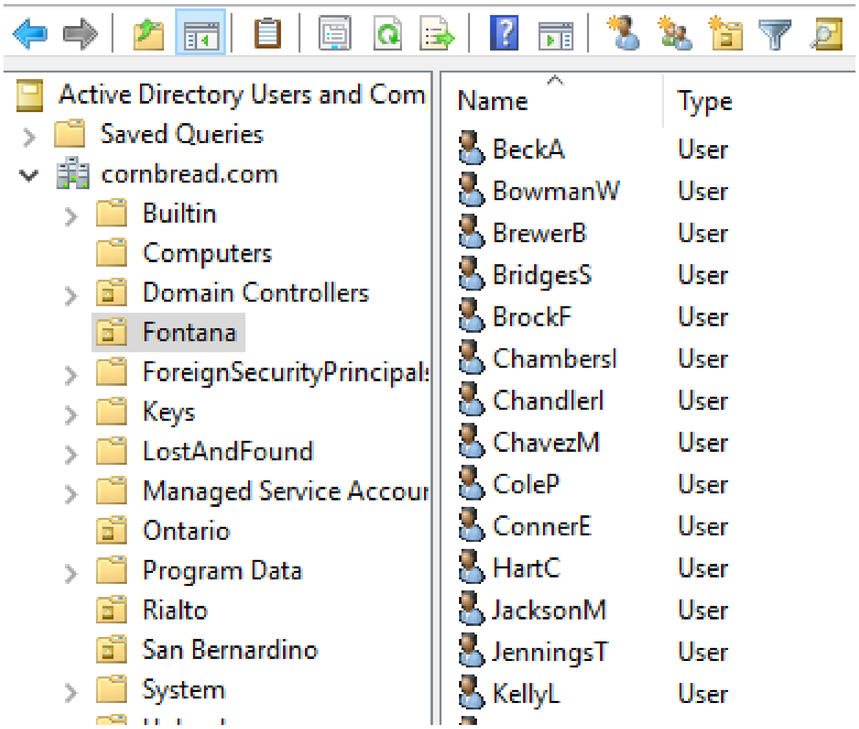
## Console Log of Users Being Created



```
$UserSyncData=Get-UserSyncData -SyncFieldMap $SyncFieldMap -UniqueID $UniqueId `
    -CSVFilePath $CsvFilePath -Delimiter $Delimiter -Domain $Domain -OUProperty $OUProperty

Create-NewUsers -UserSyncData $UserSyncData -Verbose

#New-Username -GivenName "Carl" -SurName "Johnson-Fred" -Domain "cornbread.com"

Validate-OU -SyncFieldMap $SyncFieldMap -CSVFilePath $CsvFilePath `
-Delimiter $Delimiter -Domain $Domain -OUProperty $OUProperty

#Check-Username -GivenName "Sam" -SurName "Bridges" -CurrentUsername "BeckA" -Domain "cornbread.com"

Sync-ExistingUsers -UserSyncData $UserSyncData -SyncFieldMap $SyncFieldMap -Verbose

Remove-Users -UserSyncData $UserSyncData -KeepDisabledForDays $KeepDisabledForDays -Verbose

```

```
VERBOSE: Created user: {Bill Houston} EmpID: {4092 Username: {HoustonB} Password: {qLTpL&-T=rpTrN&}
VERBOSE: Creating user : {Della Wise}
VERBOSE: Creating user : {Della Wise} with username : {WiseD}
VERBOSE: Creating user : {Della Wise} with username : {WiseD}, {OU=Ontario,DC=cornbread,DC=com)}
VERBOSE: Created user: {Della Wise} EmpID: {4093 Username: {WiseD} Password: {+vE+WqgmERTJ;06@}
VERBOSE: Creating user : {Dennis Moreno}
VERBOSE: Creating user : {Dennis Moreno} with username : {MorenoD}
VERBOSE: Creating user : {Dennis Moreno} with username : {MorenoD}, {OU=Fontana,DC=cornbread,DC=com)}
VERBOSE: Created user: {Dennis Moreno} EmpID: {4094 Username: {MorenoD} Password: {J=OK4iSf]ix!:r7oi}
VERBOSE: Creating user : {Chad Torres}
VERBOSE: Creating user : {Chad Torres} with username : {TorresC}
VERBOSE: Creating user : {Chad Torres} with username : {TorresC}, {OU=Fontana,DC=cornbread,DC=com)}
VERBOSE: Created user: {Chad Torres} EmpID: {4095 Username: {TorresC} Password: {|:.=$dnM%+IV>e)Wr}
VERBOSE: Creating user : {Alberta Chavez}
VERBOSE: Creating user : {Alberta Chavez} with username : {ChavezA}
VERBOSE: Creating user : {Alberta Chavez} with username : {ChavezA}, {OU=San Bernardino,DC=cornbread,DC=com)}
VERBOSE: Created user: {Alberta Chavez} EmpID: {4096 Username: {ChavezA} Password: {Cm@QQ}:D3S#oM}Bzk}
VERBOSE: Creating user : {Hulda Reynolds}
VERBOSE: Creating user : {Hulda Reynolds} with username : {ReynoldsH}
VERBOSE: Creating user : {Hulda Reynolds} with username : {ReynoldsH}, {OU=Rialto,DC=cornbread,DC=com)}
VERBOSE: Created user: {Hulda Reynolds} EmpID: {4097 Username: {ReynoldsH} Password: {O^S4(u3r!61Tl3=}
VERBOSE: Creating user : {Pearl Roberts}
VERBOSE: Creating user : {Pearl Roberts} with username : {RobertsP}
VERBOSE: Creating user : {Pearl Roberts} with username : {RobertsP}, {OU=Upland,DC=cornbread,DC=com)}
VERBOSE: Created user: {Pearl Roberts} EmpID: {4098 Username: {RobertsP} Password: {}ht0c{)qD11&1MJ}
VERBOSE: Creating user : {Eleanor Simmons}
VERBOSE: Creating user : {Eleanor Simmons} with username : {SimmonsE}
```

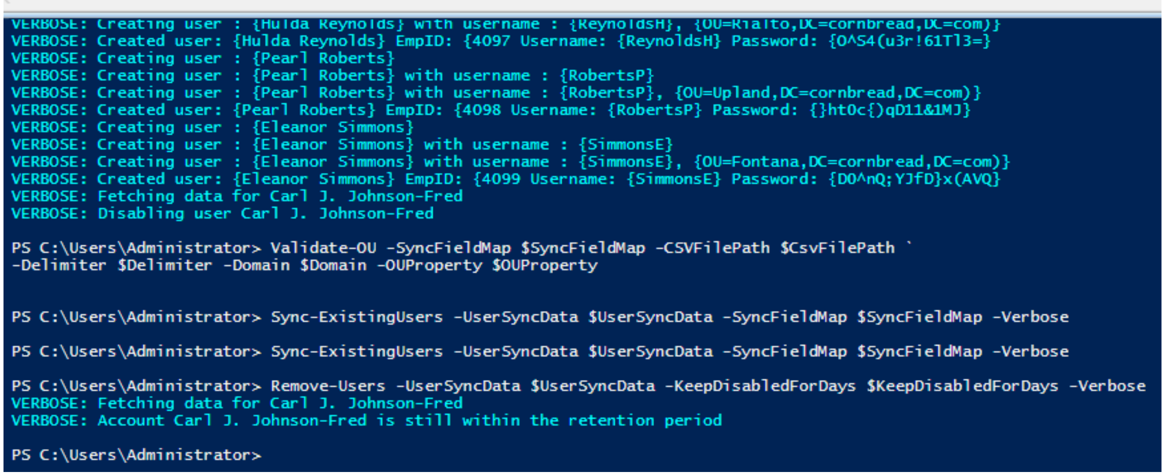
## Active Directory After



| Name | Type |
|---|---|
| BeckA | User |
| BowmanW | User |
| BrewerB | User |
| BridgesS | User |
| BrockF | User |
| ChambersI | User |
| ChandlerI | User |
| ChavezM | User |
| ColeP | User |
| ConnerE | User |
| HartC | User |
| JacksonM | User |
| JenningsT | User |
| KellyL | User |

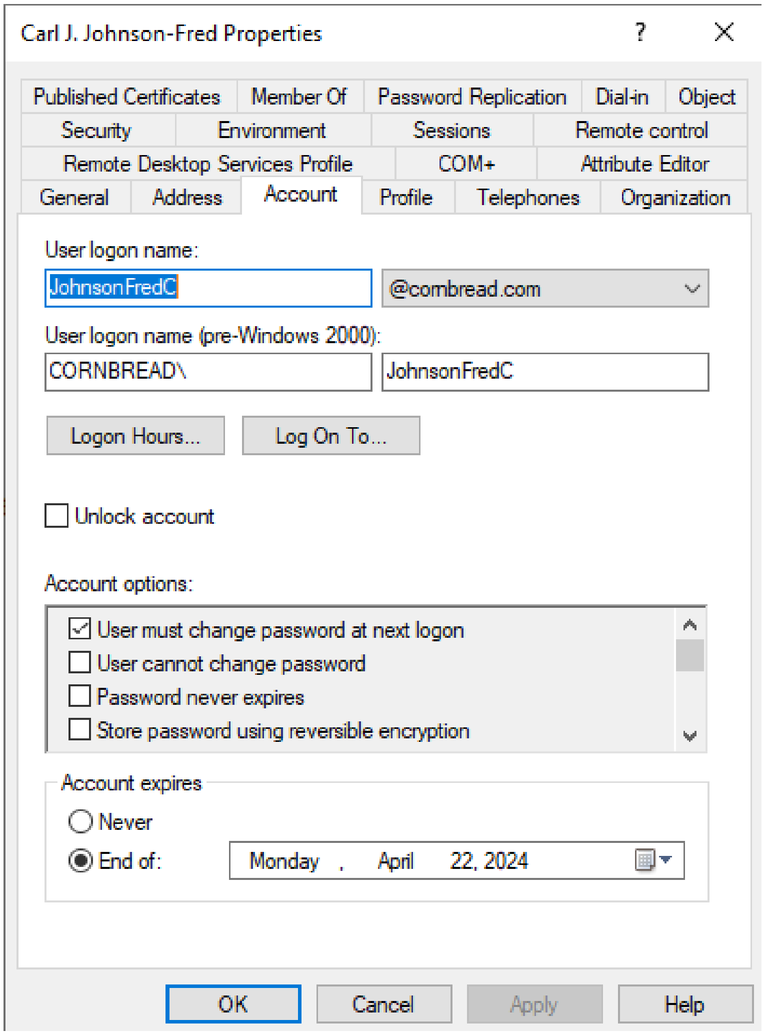
Testing Disabling and Removing

To test the account disabling and removal functionality of the script, I first ensured that the employee Carl J. Johnson-Fred was present in the CSV file. I then ran the script, which processed the CSV file and detected that Carl J. Johnson-Fred was no longer listed. The script correctly identified him as a removed user and disabled his account in the Active Directory Domain. The account was set to remain disabled for 7 days, allowing for a retention period in case the account needs to be reactivated for any reason. This test demonstrated that the script effectively handles account removals and implements a retention period before permanent deletion.



```
VERBOSE: Creating user : {Hulda Reynolds} with username : {ReynoldsH}, {OU=Rialto,DC=cornbread,DC=com)}
VERBOSE: Created user: {Hulda Reynolds} EmpID: {4097 Username: {ReynoldsH} Password: {O^S4(u3r!61Tl3=}
VERBOSE: Creating user : {Pearl Roberts}
VERBOSE: Creating user : {Pearl Roberts} with username : {RobertsP}
VERBOSE: Creating user : {Pearl Roberts} with username : {RobertsP}, {OU=Upland,DC=cornbread,DC=com)}
VERBOSE: Created user: {Pearl Roberts} EmpID: {4098 Username: {RobertsP} Password: {}ht0c{)qD11&1MJ}
VERBOSE: Creating user : {Eleanor Simmons}
VERBOSE: Creating user : {Eleanor Simmons} with username : {SimmonsE}
VERBOSE: Creating user : {Eleanor Simmons} with username : {SimmonsE}, {OU=Fontana,DC=cornbread,DC=com)}
VERBOSE: Created user: {Eleanor Simmons} EmpID: {4099 Username: {SimmonsE} Password: {DO^nQ;YJfD}x(AVQ}
VERBOSE: Fetching data for Carl J. Johnson-Fred
VERBOSE: Disabling user Carl J. Johnson-Fred

PS C:\Users\Administrator> Validate-OU -SyncFieldMap $SyncFieldMap -CSVFilePath $CsvFilePath `
-Delimiter $Delimiter -Domain $Domain -OUProperty $OUProperty


PS C:\Users\Administrator> Sync-ExistingUsers -UserSyncData $UserSyncData -SyncFieldMap $SyncFieldMap -Verbose

PS C:\Users\Administrator> Sync-ExistingUsers -UserSyncData $UserSyncData -SyncFieldMap $SyncFieldMap -Verbose

PS C:\Users\Administrator> Remove-Users -UserSyncData $UserSyncData -KeepDisabledForDays $KeepDisabledForDays -Verbose
VERBOSE: Fetching data for Carl J. Johnson-Fred
VERBOSE: Account Carl J. Johnson-Fred is still within the retention period

PS C:\Users\Administrator>
```

Account Expiring in Seven Days



User logon name: JohnsonFredC @cornbread.com

User logon name (pre-Windows 2000): CORNBREAD\ JohnsonFredC

□ Unlock account

Account options:

- ☑ User must change password at next logon
- □ User cannot change password
- □ Password never expires
- □ Store password using reversible encryption

Account expires

- ○ Never
- ● End of: Monday , April 22, 2024

## Conclusion, Lessons Learned, & Plans for the Future

In conclusion, this PowerShell script represents a significant milestone in my programming journey. It has not only taught me how to use PowerShell ISE effectively but also introduced me to new coding strategies. Through this project, I have gained valuable insights into the potential for automation and improvement in managing Active Directory. Compared to my initial project involving virtual machines, this experience has been a substantial step forward in building my overall IT knowledge foundation. Projects like these highlight the importance of hands-on experience in the IT field. They provide a practical understanding of complex systems like Active Directory and demonstrate the power of automation in simplifying administrative tasks. This project, in particular, has shown me the potential for further automation and enhancement in managing Active Directory, leaving me eager to explore more advanced scripting techniques and expand my IT skills further.