



Exercise - Access secrets stored in Azure Key Vault

13 minutes

Sandbox activated! Time remaining: 3 hr 39 min

You have used 2 of 10 sandboxes for today. More sandboxes will be available tomorrow.

Choose your development language

C#

JavaScript

Now that you know how enabling managed identities for Azure resources creates an identity for our app to use for authentication, we'll create an app that uses that identity to access secrets in the vault.

Reading secrets in a Node.js app

The Azure Key Vault API is a REST API that handles all management and usage of keys and vaults. Each secret in a vault has a unique URL, and secret values are retrieved with HTTP GET requests.

The official Key Vault client for Node.js apps is the `KeyVaultClient` class in the `azure-keyvault` npm package. Apps that include secret names in their configuration or code will generally only need to use its `getSecret` method, which loads a secret value given its name. `getSecret` requires your app's identity to have the **Get** permission on the vault. Apps designed to load all secrets from a vault will also use the `getSecrets` method, which loads a list of secrets and requires the **List** permission.

Before your app can create a `KeyVaultClient` instance, it must get a credential object by authenticating to the vault. To authenticate, use the one of the login functions provided by the `ms-rest-azure` npm package. Each of these functions will return a credential object that can be used to create a `KeyVaultClient`. The `loginWithAppServiceMSI` function will automatically use the managed identity credentials that App Service makes available to your app via environment variables. For test environments or other non-App Service environments where

your app does not have access to a managed identity, you can manually create a service principal for your app and use the `loginWithServicePrincipalSecret` function to authenticate.

Tip

Regardless of the framework or language you use to build your app, you should design it to cache secret values locally or load them into memory at startup unless you have a specific reason not to. Reading them directly from the vault every time you need them is unnecessarily slow and expensive.

Handling secrets in an app

Once a secret is loaded into your app, it's up to your app to handle it securely. In the app we build in this module, we write our secret value out to the client response and view it in a web browser to demonstrate that it has been loaded successfully. **Returning a secret value to the client is *not* something you'd normally do!** Usually, you'll use secrets to do things like initialize client libraries for databases or remote APIs.

Important

Always carefully review your code to ensure that your app never writes secrets to any kind of output, including logs, storage, and responses.

Exercise

We'll create a new web API with Express.js and use the `azure-keyvault` and `ms-rest-azure` packages to load the secret from our vault.

Create the app

In the Azure Cloud Shell terminal, run the following to initialize a new Node.js application, install the needed packages, and open a new file in the editor.

console

 Copy

```
mkdir KeyVaultDemoApp
cd KeyVaultDemoApp
npm init -y
npm install ms-rest-azure azure-keyvault express
touch app.js
code app.js
```

Add code to load and use secrets

To demonstrate good usage of Key Vault, our app will load secrets from the vault at startup. To demonstrate that our secrets have been loaded, we'll create an endpoint that displays the value of the **SecretPassword** secret.

First, paste the following code into the editor to set up the application. This will import the necessary packages, set up the port and vault URL configuration, and create a new object to hold the secret names and values.

JavaScript

 Copy



```
// Importing dependencies
const msRestAzure = require('ms-rest-azure');
const keyVault = require('azure-keyvault');
const app = require('express')();

// Initialize port
const port = process.env.PORT || 3000;

// Create Vault URL from App Settings
const vaultUrl = `https://${process.env.VaultName}.vault.azure.net/`;

// Map of key vault secret names to values
let vaultSecretsMap = {};
```

Important

Make sure to save files as you work on them, especially when you're finished. You can do this either through the "..." menu, or the accelerator key ( on Windows and Linux,  on macOS).

Next, we'll add the code to authenticate to the vault and load the secrets. We'll add this as two separate functions. Insert a couple of blank lines after the code you previously added and then paste in the following code:

JavaScript

 Copy

```
const authenticateToKeyVault = async () => {
  try {
    let credentials;
    if (process.env.NODE_ENV === 'production') {
      credentials = await msRestAzure.loginWithAppServiceMSI({ resource:
'https://vault.azure.net' });
    } else {
      // For non-App Service environments. Set the APP_ID, APP_SECRET and
      TENANT_ID environment
```

```

    // variables to use.
    const appId = process.env.APP_ID;
    const appSecret = process.env.APP_SECRET;
    const tenantId = process.env.TENANT_ID;
    credentials = await msRestAzure.loginWithServicePrincipalSecret(appId,
appSecret, tenantId);
  }
  return credentials;
} catch(err) {
  throw err.message;
}
}

const getKeyVaultSecrets = async credentials => {
  // Create a key vault client
  let keyVaultClient = new keyVault.KeyVaultClient(credentials);
  try {
    let secrets = await keyVaultClient.getSecrets(vaultUrl);
    // For each secret name, get the secret value from the vault
    for (const secret of secrets) {
      // Only load enabled secrets - getSecret will return an error for disabled
secrets
      if (secret.attributes.enabled) {
        let secretId = secret.id;
        let secretName = secretId.substring(secretId.lastIndexOf('/') + 1);
        let secretValue = await keyVaultClient.getSecret(vaultUrl, secretName,
'');
        vaultSecretsMap[secretName] = secretValue.value;
      }
    }
  } catch(err) {
    console.log(err.message)
  }
}

```

Now create the Express endpoint we'll use to test whether our secret was loaded. Paste in this code next:

JavaScript

 Copy

```

app.get('/api/SecretTest', (req, res) => {
  let secretName = 'SecretPassword';
  let response;
  if (secretName in vaultSecretsMap) {
    response = `Secret value: ${vaultSecretsMap[secretName]}\n\nThis is for
testing only! Never output a secret to a response or anywhere else in a real
app!`;
  } else {
    response = `Error: No secret named ${secretName} was found...`
  }
  res.type('text');
  res.send(response);
});

```

Finally, we'll call our functions to load the secrets from our vault, then start the app. Paste in this last snippet to complete the application:


JavaScript

 Copy

```
(async () => {  
  let credentials = await authenticateToKeyVault();  
  await getKeyVaultSecrets(credentials);  
  app.listen(port, () => {  
    console.log(`Server running at http://localhost:${port}`);  
  });  
})();  
}).catch(err => console.log(err));
```

We're finished writing code, so make sure to save the file. The app is ready to run — now let's get it into Azure!

Next unit: Exercise - Configure, deploy, and run in Azure

[Continue >](#) English (United States)

[Previous Version Docs](#) • [Blog](#) • [Contribute](#) • [Privacy & Cookies](#) • [Terms of Use](#) • [Trademarks](#) •

© Microsoft 2020

 Azure Cloud Shell

app.js		...
64	res.type('text');	
65	res.send(response);	

```
rajani_net@Azure:~/KeyVaultDemoApp$ code app
rajani_net@Azure:~/KeyVaultDemoApp$ node app
clientId must be a non empty string.
rajani_net@Azure:~/KeyVaultDemoApp$
```