Package 'AzureKeyVault'

October 12, 2020

Title Key and Secret Management in 'Azure'

Version 1.0.4

Description Manage keys, certificates, secrets, and storage accounts in Microsoft's 'Key Vault' service: <https://azure.microsoft.com/services/key-vault/>. Provides facilities to store and retrieve secrets, use keys to encrypt, decrypt, sign and verify data, and manage certificates. Integrates with the 'AzureAuth' package to enable authentication with a certificate, and with the 'openssl' package for importing and exporting cryptographic objects. Part of the 'AzureR' family of packages.

License MIT + file LICENSE

URL https://github.com/Azure/AzureKeyVault
https://github.com/Azure/AzureR

BugReports https://github.com/Azure/AzureKeyVault/issues

VignetteBuilder knitr

Depends R (>= 3.3),

Imports utils, R6, httr, jsonlite, openssl, jose, AzureRMR, AzureGraph, AzureAuth (>= 1.0.1)

Suggests AzureStor, knitr, rmarkdown, testthat

RoxygenNote 6.1.1

NeedsCompilation no

Author Hong Ooi [aut, cre], Microsoft [cph]

Maintainer Hong Ooi <hongooi73@gmail.com>

Repository CRAN

Date/Publication 2020-10-12 11:30:02 UTC

R topics documented:

`AzureKeyVault` *Azure Key Vault endpoint class*

Description

Class representing the client endpoint for a key vault, exposing methods for working with it. Use the `[key_vault]` function to instantiate new objects of this class.

Usage

```
AzureKeyVault
```

Format

An object of class `R6ClassGenerator` of length 25.

Fields

- `keys`: A sub-object for working with encryption keys stored in the vault. See keys.
- `secrets`: A sub-object for working with secrets stored in the vault. See secrets.
- `certificates`: A sub-object for working with certificates stored in the vault. See certificates.
- `storage`: A sub-object for working with storage accounts managed by the vault. See storage.

See Also

key_vault, keys, secrets, certificates, storage

Azure Key Vault documentation, Azure Key Vault API reference

Examples

```
## Not run:

key_vault("mykeyvault")
key_vault("https://mykeyvault.vault.azure.net")

# authenticating as a service principal
key_vault("mykeyvault", tenant="myaadtenant", app="app_id", password="password")

# authenticating with an existing token
token <- AzureAuth::get_azure_token("https://vault.azure.net", "myaadtenant",
                                    app="app_id", password="password")
key_vault("mykeyvault", token=token)


## End(Not run)
```

---

az_key_vault *Key vault resource class*

Description

Class representing a key vault, exposing methods for working with it.

Usage

```
az_key_vault
```

Format

An object of class `R6ClassGenerator` of length 25.

Methods

The following methods are available, in addition to those provided by the AzureRMR::az_resource class:

- `new(...)`: Initialize a new key vault object. See 'Initialization'.
- `add_principal(principal,...)`: Add an access policy for a user or service principal. See 'Access policies' below.
- `get_principal(principal)`: Retrieve an access policy for a user or service principal.
- `remove_principal(principal)`: Remove access for a user or service principal.
- `get_endpoint()`: Return the vault endpoint. See 'Endpoint' below.

Initialization

Initializing a new object of this class can either retrieve an existing key vault, or create a new vault on the host. The recommended way to initialize an object is via the `get_key_vault`, `create_key_vault` or `list_key_vaults` methods of the az_resource_group class, which handle the details automatically.

Access policies

Client access to a key vault is governed by its access policies, which are set on a per-principal basis. Each principal (user or service) can have different permissions granted, for keys, secrets, certificates, and storage accounts.

To grant access, use the `add_principal` method. This has signature

```
add_principal(principal, tenant = NULL,
              key_permissions = "all",
              secret_permissions = "all",
              certificate_permissions = "all",
              storage_permissions = "all")
```

The `principal` can be a GUID, an object of class `vault_access_policy`, or a user, app or service principal object from the AzureGraph package. Note that the app ID of a registered app is not the same as the ID of its service principal.

The tenant must be a GUID; if this is NULL, it will be taken from the tenant of the key vault resource.

Here are the possible permissions for keys, secrets, certificates, and storage accounts. The permission "all" means to grant all permissions.

- Keys: "get", "list", "update", "create", "import", "delete", "recover", "backup", "restore", "decrypt", "encrypt", "unwrapkey", "wrapkey", "verify", "sign", "purge"
- Secrets: "get", "list", "set", "delete", "recover", "backup", "restore", "purge"
- Certificates: "get", "list", "update", "create", "import", "delete", "recover", "backup", "restore", "managecontacts", "manageissuers", "getissuers", "listissuers", "setissuers", "deleteissuers", "purge"
- Storage accounts: "get", "list", "update", "set", "delete", "recover", "backup", "restore", "regeneratekey", "getsas", "listsas", "setsas", "deletesas", "purge"

To revoke access, use the `remove_principal` method. To view the current access policy, use `get_principal` or `list_principals`.

Endpoint

The client-side interaction with a key vault is via its *endpoint*, which is usually at the URL `https://[vaultname].vault.azu`
The `get_endpoint` method returns an R6 object of class `key_vault`, which represents the endpoint. Authenticating with the endpoint is done via an OAuth token; the necessary credentials are taken from the current Resource Manager client in use, or you can supply your own.

```
get_endpoint(tenant = self$token$tenant,
             app = self$token$client$client_id,
             password = self$token$client$client_secret, ...)
```

To access the key vault independently of Resource Manager (for example if you are a user without admin or owner access to the vault resource), use the key_vault function.

See Also

vault_access_policy, key_vault create_key_vault, get_key_vault, delete_key_vault, AzureGraph::get_graph_login, AzureGraph::az_user, AzureGraph::az_app, AzureGraph::az_service_principal

Azure Key Vault documentation, Azure Key Vault API reference

Examples

```
## Not run:

# recommended way of retrieving a resource: via a resource group object
kv <- resgroup$get_key_vault("mykeyvault")

# list principals that have access to the vault
kv$list_principals()

# grant a user full access (the default)
usr <- AzureGraph::get_graph_login()$
    get_user("username@aadtenant.com")
kv$add_principal(usr)

# grant a service principal read access to keys and secrets only
svc <- AzureGraph::get_graph_login()$
    get_service_principal(app_id="app_id")
kv$add_principal(svc,
    key_permissions=c("get", "list"),
    secret_permissions=c("get", "list"),
    certificate_permissions=NULL,
    storage_permissions=NULL)
# alternatively, supply a vault_access_policy with the listed permissions
pol <- vault_access_policy(svc,
    key_permissions=c("get", "list"),
    secret_permissions=c("get", "list"),
    certificate_permissions=NULL,
    storage_permissions=NULL)
kv$add_principal(pol)

# revoke access
kv$remove_access(svc)

# get the endpoint object
vault <- kv$get_endpoint()


## End(Not run)
```

`certificate` *Certificate object*

Description

This class represents a certificate stored in a vault. It provides methods for carrying out operations, including encryption and decryption, signing and verification, and wrapping and unwrapping.

Fields

This class provides the following fields:

- `cer`: The contents of the certificate, in CER format.
- `id`: The ID of the certificate.
- `kid`: The ID of the key backing the certificate.
- `sid`: The ID of the secret backing the certificate.
- `contentType`: The content type of the secret backing the certificate.
- `policy`: The certificate management policy, containing the authentication details.
- `x5t`: The thumbprint of the certificate.

Methods

This class provides the following methods:

```
export(file)
export_cer(file)
sign(digest, ...)
verify(signature, digest, ...)
set_policy(subject=NULL, x509=NULL, issuer=NULL,
           key=NULL, secret_type=NULL, actions=NULL,
           attributes=NULL, wait=TRUE)
get_policy()
sync()

update_attributes(attributes=vault_object_attrs(), ...)
list_versions()
set_version(version=NULL)
delete(confirm=TRUE)
```

Arguments

- `file`: For `export` and `export_cer`, a connection object or a character string naming a file to export to.
- `digest`: For `sign`, a hash digest string to sign. For `verify`, a digest to compare to a signature.
- `signature`: For `verify`, a signature string.

- `subject,x509,issuer,key,secret_type,actions,wait`: These are the same arguments as used when creating a new certificate. See certificates for more information.
- `attributes`: For `update_attributes`, the new attributes for the object, such as the expiry date and activation date. A convenient way to provide this is via the vault_object_attrs helper function.
- `...`: For `update_attributes`, additional key-specific properties to update. For `sign` and `verify`, additional arguments for the corresponding key object methods. See keys and key.
- `version`: For `set_version`, the version ID or NULL for the current version.
- `confirm`: For `delete`, whether to ask for confirmation before deleting the key.

Details

`export` exports the full certificate to a file. The format wll be either PEM or PFX (aka PKCS#12), as set by the `format` argument when the certificate was created. `export_cer` exports the public key component, aka the CER file. Note that the public key can also be found in the `cer` field of the object.

`sign` uses the key associated with the a certificate to sign a digest, and `verify` checks a signature against a digest for authenticity. See below for an example of using `sign` to do OAuth authentication with certificate credentials.

`set_policy` updates the authentication details of a certificate: its issuer, identity, key type, renewal actions, and so on. `get_policy` returns the current policy of a certificate.

A certificate can have multiple *versions*, which are automatically generated when a cert is created with the same name as an existing cert. By default, this object contains the information for the most recent (current) version; use `list_versions` and `set_version` to change the version.

Value

For `get_policy`, a list of certificate policy details.

For `list_versions`, a data frame containing details of each version.

For `set_version`, the key object with the updated version.

See Also

certificates

Azure Key Vault documentation, Azure Key Vault API reference

Examples

```
## Not run:

vault <- key_vault("mykeyvault")

cert <- vault$certificates$create("mynewcert")
cert$cer
cert$export("mynewcert.pem")

# new version of an existing certificate
```

```
vault$certificates$create("mynewcert", x509=cert_x509_properties(validity_months=24))

cert <- vault$certificates$get("mynewcert")
vers <- cert$list_versions()
cert$set_version(vers[2])

# updating an existing cert version
cert$set_policy(x509=cert_x509_properties(validity_months=12))


## signing a JSON web token (JWT) for authenticating with Azure Active Directory
app <- "app_id"
tenant <- "tenant_id"
claim <- jose::jwt_claim(
    iss=app,
    sub=app,
    aud="https://login.microsoftonline.com/tenant_id/oauth2/token",
    exp=as.numeric(Sys.time() + 60*60),
    nbf=as.numeric(Sys.time())
)
# header includes cert thumbprint
header <- list(alg="RS256", typ="JWT", x5t=cert$x5t)

token_encode <- function(x)
{
    jose::base64url_encode(jsonlite::toJSON(x, auto_unbox=TRUE))
}
token_contents <- paste(token_encode(header), token_encode(claim), sep=".")

# get the signature and concatenate it with header and claim to form JWT
sig <- cert$sign(openssl::sha256(charToRaw(token_contents)))
cert_creds <- paste(token_contents, sig, sep=".")

AzureAuth::get_azure_token("resource_url", tenant, app, certificate=cert_creds)


## End(Not run)
```

`certificates` *Certificates in Key Vault*

Description

This class represents the collection of certificates stored in a vault. It provides methods for managing certificates, including creating, importing and deleting certificates, and doing backups and restores. For operations with a specific certificate, see certificate.

Methods

This class provides the following methods:

```
create(name, subject, x509=cert_x509_properties(), issuer=cert_issuer_properties(),
       key=cert_key_properties(), format=c("pem", "pkcs12"),
       expiry_action=cert_expiry_action(),
       attributes=vault_object_attrs(),
       ..., wait=TRUE)
import(name, value, pwd=NULL,
       attributes=vault_object_attrs(),
       ..., wait=TRUE)
get(name)
delete(name, confirm=TRUE)
list()
backup(name)
restore(backup)
get_contacts()
set_contacts(email)
add_issuer(issuer, provider, credentials=NULL, details=NULL)
remove_issuer(issuer)
get_issuer(issuer)
list_issuers()
```

Arguments

- `name`: The name of the certificate.
- `subject`: For `create`, The subject or X.500 distinguished name for the certificate.
- `x509`: Other X.509 properties for the certificate, such as the domain name(s) and validity period. A convenient way to provide this is via the cert_x509_properties helper function.
- `issuer`: Issuer properties for the certificate. A convenient way to provide this is via the cert_issuer_properties helper function. The default is to specify a self-signed certificate.
- `key`: Key properties for the certificate. A convenient way to provide this is via the cert_key_properties helper function.
- `format`: The format to store the certificate in. Can be either PEM or PFX, aka PKCS#12. This also determines the format in which the certificate will be exported (see certificate).
- `expiry_action`: What Key Vault should do when the certificate is about to expire. A convenient way to provide this is via the cert_expiry_action helper function.
- `attributes`: Optional attributes for the secret. A convenient way to provide this is via the vault_object_attrs helper function.
- `value`: For `import`, the certificate to import. This can be the name of a PFX file, or a raw vector with the contents of the file.
- `pwd`: For `import`, the password if the imported certificate is password-protected.
- `...`: For `create` and `import`, other named arguments which will be treated as tags.
- `wait`: For `create` and `import`, whether to wait until the certificate has been created before returning. If FALSE, you can check on the status of the certificate via the returned object's `sync` method.
- `backup`: For `restore`, a string representing the backup blob for a key.
- `email`: For `set_contacts`, the email addresses of the contacts.

- `issuer`: For the issuer methods, the name by which to refer to an issuer.
- `provider`: For `add_issuer`, the provider name as a string.
- `credentials`: For `add_issuer`, the credentials for the issuer, if required. Should be a list containing the components `account_id` and `password`.
- `details`: For `add_issuer`, the organisation details, if required. See the Azure docs for more information.

Value

For `get`, `create` and `import`, an object of class `stored_certificate`, representing the certificate itself.

For `list`, a vector of key names.

For `add_issuer` and `get_issuer`, an object representing an issuer. For `list_issuers`, a vector of issuer names.

For `backup`, a string representing the backup blob for a certificate. If the certificate has multiple versions, the blob will contain all versions.

See Also

certificate, cert_key_properties, cert_x509_properties, cert_issuer_properties, vault_object_attrs

Azure Key Vault documentation, Azure Key Vault API reference

Examples

```
## Not run:

vault <- key_vault("mykeyvault")

vault$certificates$create("mynewcert", "CN=mydomain.com")
vault$certificates$list()
vault$certificates$get("mynewcert")

# specifying some domain names
vault$certificates$create("mynewcert", "CN=mydomain.com",
    x509=cert_x509_properties(dns_names=c("mydomain.com", "otherdomain.com")))

# specifying a validity period of 2 years (24 months)
vault$certificates$create("mynewcert", "CN=mydomain.com",
    x509=cert_x509_properties(validity_months=24))

# setting management tags
vault$certificates$create("mynewcert", "CN=mydomain.com", tag1="a value", othertag="another value")

# importing a cert from a PFX file
vault$certificates$import("importedcert", "mycert.pfx")

# backup and restore a cert
bak <- vault$certificates$backup("mynewcert")
vault$certificates$delete("mynewcert", confirm=FALSE)
```

```
vault$certificates$restore(bak)

# set a contact
vault$certificates$set_contacts("username@mydomain.com")
vault$certificates$get_contacts()

# add an issuer and then obtain a cert
# this can take a long time, so set wait=FALSE to return immediately
vault$certificates$add_issuer("newissuer", provider="OneCert")
vault$certificates$create("issuedcert", "CN=mydomain.com",
    issuer=cert_issuer_properties("newissuer"),
    wait=FALSE)


## End(Not run)
```

`cert_key_properties` *Helper functions for key vault objects*

Description

Helper functions for key vault objects

Usage

```
cert_key_properties(type = c("RSA", "EC"), hardware = FALSE,
  ec_curve = NULL, rsa_key_size = NULL, key_exportable = TRUE,
  reuse_key = FALSE)

cert_x509_properties(dns_names = character(), emails = character(),
  upns = character(), key_usages = c("digitalSignature",
  "keyEncipherment"), enhanced_key_usages = c("1.3.6.1.5.5.7.3.1",
  "1.3.6.1.5.5.7.3.2"), validity_months = NULL)

cert_issuer_properties(issuer = "self", cert_type = NULL,
  transparent = NULL)

cert_expiry_action(remaining = 0.1, action = c("AutoRenew",
  "EmailContacts"))

vault_object_attrs(enabled = TRUE, expiry_date = NULL,
  activation_date = NULL, recovery_level = NULL)
```

Arguments

`type`	For `cert_key_properties`, the type of key to create: RSA or elliptic curve (EC). Note that for keys backing a certificate, only RSA is allowed.

`hardware` For `cert_key_properties`, whether to use a hardware key or software key. The former requires a premium key vault.

`ec_curve` For an EC key, the type of elliptic curve.

`rsa_key_size` For an RSA key, the key size, either 2048, 3072 or 4096.

`key_exportable` For a key used in a certificate, whether it should be exportable.

`reuse_key` For a key used in a certificate, whether it should be reused when renewing the certificate.

`dns_names, emails, upns`
For `cert_x509_properties`, the possible subject alternative names (SANs) for a certificate. These should be character vectors.

`key_usages` For `cert_x509_properties`, a character vector of key usages.

`enhanced_key_usages`
For `cert_x509_properties`, a character vector of enhanced key usages (EKUs).

`validity_months`
For `cert_x509_properties`, the number of months the certificate should be valid for.

`issuer` For `cert_issuer_properties`, the name of the issuer. Defaults to "self" for a self-signed certificate.

`cert_type` For `cert_issuer_properties`, the type of certificate to issue, eg "OV-SSL", "DV-SSL" or "EV-SSL".

`transparent` For `cert_issuer_properties`, whether the certificate should be transparent.

`remaining` For `cert_expiry_action`, The remaining certificate lifetime at which to take action. If this is a number between 0 and 1, it is interpreted as the percentage of life remaining; otherwise, the number of days remaining. To disable expiry actions, set this to NULL.

`action` For `cert_expiry_action`, what action to take when a certificate is about to expire. Can be either "AutoRenew" or "EmailContacts". Ignored if `remaining == NULL`.

`enabled` For `vault_object_attrs`, whether this stored object (key, secret, certificate, storage account) is enabled.

`expiry_date, activation_date`
For `vault_object_attrs`, the optional expiry date and activation date of the stored object. Can be any R object that can be coerced to POSIXct format.

`recovery_level` For `vault_object_attrs`, the recovery level for the stored object.

Details

These are convenience functions for specifying the properties of objects stored in a key vault. They return lists of fields to pass to the REST API.

`create_key_vault` *Create Azure key vault*

Description

Method for the AzureRMR::az_resource_group class.

Usage

```
create_key_vault(name, location = self$location, initial_access = default_access(),
                 sku = "Standard", ..., wait = TRUE)
```

Arguments

- `name`: The name of the key vault.
- `location`: The location/region in which to create the account. Defaults to the resource group location.
- `initial_access`: The user or service principals that will have access to the vault. This should be a list of objects of type `[vault_access_policy]`, created by the function of the same name. The default is to grant access to the logged-in user or service principal of the current Resource Manager client.
- `sku`: The sku for the vault. Set this to "Premium" to enable the use of hardware security modules (HSMs).
- `allow_vm_access`: Whether to allow Azure virtual machines to retrieve certificates from the vault.
- `allow_arm_access`: Whether to allow Azure Resource Manager to retrieve secrets from the vault for template deployment purposes.
- `allow_disk_encryption_access`: Whether to allow Azure Disk Encryption to retrieve secrets and keys from the vault.
- `soft_delete`: Whether soft-deletion should be enabled for this vault. Soft-deletion is a feature which protects both the vault itself and its contents from accidental/malicious deletion; see below.
- `purge_protection`: Whether purge protection is enabled. If this is TRUE and soft-deletion is enabled for the vault, manual purges are not allowed. Has no effect if `soft_delete=FALSE`.
- `...`: Other named arguments to pass to the az_key_vault initialization function.
- `wait`: Whether to wait for the resource creation to complete before returning.

Details

This method deploys a new key vault resource, with parameters given by the arguments. A key vault is a secure facility for storing and managing encryption keys, certificates, storage account keys, and generic secrets.

A new key vault will have access granted to the user or service principal used to sign in to the Azure Resource Manager client. To manage access policies after creation, use the `add_principal`, `list_principals` and `remove_principal` methods of the key vault object.

Key Vault's soft delete feature allows recovery of the deleted vaults and vault objects, known as soft-delete. Specifically, it addresses the following scenarios:

- Support for recoverable deletion of a key vault
- Support for recoverable deletion of key vault objects (keys, secrets, certificates)

With this feature, the delete operation on a key vault or key vault object is a soft-delete, effectively holding the resources for a given retention period (90 days), while giving the appearance that the object is deleted. The service further provides a mechanism for recovering the deleted object, essentially undoing the deletion.

Soft-deleted vaults can be purged (permanently removed) by calling the `purge_key_vault` method for the resource group or subscription classes. The purge protection optional feature provides an additional layer of protection by forbidding manual purges; when this is on, a vault or an object in deleted state cannot be purged until the retention period of 90 days has passed.

To see what soft-deleted key vaults exist, call the `list_deleted_key_vaults` method. To recover a soft-deleted key vault, call the `create_key_vault` method from the vault's original resource group, with the vault name. To purge (permanently delete) it, call the `purge_key_vault` method.

Value

An object of class `az_key_vault` representing the created key vault.

See Also

get_key_vault, delete_key_vault, purge_key_vault, az_key_vault, vault_access_policy

Azure Key Vault documentation, Azure Key Vault API reference

Examples

```
## Not run:

rg <- AzureRMR::get_azure_login()$
    get_subscription("subscription_id")$
    get_resource_group("rgname")

# create a new key vault
rg$create_key_vault("mykeyvault")

# create a new key vault, and grant access to a service principal
gr <- AzureGraph::get_graph_login()
svc <- gr$get_service_principal("app_id")
rg$create_key_vault("mykeyvault",
    initial_access=list(vault_access_policy(svc, tenant=NULL)))


## End(Not run)
```

`delete_key_vault` *Delete an Azure Key Vault*

Description

Method for the AzureRMR::az_resource_group class.

Details

Deleting a key vault that has soft-deletion enabled does not permanently remove it. Instead the resource is held for a given retention period (90 days), during which it can be recovered, essentially undoing the deletion.

To see what soft-deleted key vaults exist, call the `list_deleted_key_vaults` method. To recover a soft-deleted key vault, call the `create_key_vault` method from the vault's original resource group, with the vault name. To purge (permanently delete) it, call the `purge_key_vault` method.

Usage

```
delete_key_vault(name, confirm=TRUE, wait=FALSE, purge=FALSE)
```

Arguments

- `name`: The name of the key vault.
- `confirm`: Whether to ask for confirmation before deleting.
- `wait`: Whether to wait until the deletion is complete. Note that `purge=TRUE` will set `wait=TRUE` as well.
- `purge`: For a vault with the soft-deletion feature enabled, whether to purge it as well (hard delete). Has no effect if the vault does not have soft-deletion enabled.

Value

NULL on successful deletion.

See Also

create_key_vault, get_key_vault, purge_key_vault, list_deleted_key_vaults, az_key_vault,

Azure Key Vault documentation, Azure Key Vault API reference

Examples

```
## Not run:

rg <- AzureRMR::get_azure_login()$
    get_subscription("subscription_id")$
    get_resource_group("rgname")

# assuming the vault has soft-delete enabled
```

```
rg$delete_key_vault("mykeyvault", purge=FALSE)

# recovering a soft-deleted key vault
rg$create_key_vault("mykeyvault")

# deleting it for good
rg$delete_key_vault("mykeyvault", purge=FALSE)


## End(Not run)
```

`get_key_vault` *Get existing Azure Key Vault*

Description

Methods for the AzureRMR::az_resource_group class.

Usage

```
get_key_vault(name)
list_key_vaults()
```

Arguments

- `name`: For `get_key_vault()`, the name of the key vault.

Value

For `get_key_vault()`, an object of class `az_key_vault` representing the vault.

For `list_key_vaults()`, a list of such objects.

See Also

create_key_vault, delete_key_vault, az_key_vault

Azure Key Vault documentation, Azure Key Vault API reference

Examples

```
## Not run:

rg <- AzureRMR::get_azure_login()$
    get_subscription("subscription_id")$
    get_resource_group("rgname")

rg$list_key_vaults()

rg$get_key_vault("mykeyvault")
```

```
## End(Not run)
```

---

key *Encryption key object*

Description

This class represents an encryption key stored in a vault. It provides methods for carrying out operations, including encryption and decryption, signing and verification, and wrapping and unwrapping.

Fields

This class provides the following fields:

- `key`: The key details as a parsed JSON web key (JWK).
- `managed`: Whether this key's lifetime is managed by Key Vault. TRUE if the key backs a certificate.

Methods

This class provides the following methods:

```
encrypt(plaintext, algorithm=c("RSA-OAEP", "RSA-OAEP-256", "RSA1_5"))
decrypt(ciphertext, algorithm=c("RSA-OAEP", "RSA-OAEP-256", "RSA1_5"), as_raw=TRUE)
sign(digest,
     algorithm=c("RS256", "RS384", "RS512", "PS256", "PS384", "PS512",
                 "ES256", "ES256K", "ES384", "ES512"))
verify(signature, digest,
       algorithm=c("RS256", "RS384", "RS512", "PS256", "PS384", "PS512",
                   "ES256", "ES256K", "ES384", "ES512"))
wrap(value, algorithm=c("RSA-OAEP", "RSA-OAEP-256", "RSA1_5"))
unwrap(value, algorithm=c("RSA-OAEP", "RSA-OAEP-256", "RSA1_5"), as_raw=TRUE)

update_attributes(attributes=vault_object_attrs(), ...)
list_versions()
set_version(version=NULL)
delete(confirm=TRUE)
```

Arguments

- `plaintext`: For `encrypt`, the plaintext to encrypt.
- `ciphertext`: For `decrypt`, the ciphertext to decrypt.
- `digest`: For `sign`, a generated hash to sign. For `verify`, the digest to verify for authenticity.
- `signature`: For `verify`, a signature to verify for authenticity.
- `value`: For `wrap`, a symmetric key to be wrapped; for `unwrap`, the value to be unwrapped to obtain the symmetric key.

- as_raw: For decrypt and unwrap, whether to return a character vector or a raw vector (the default).
- algorithm: The algorithm to use for each operation. Note that the algorithm must be compatible with the key type, eg RSA keys cannot use ECDSA for signing or verifying.
- attributes: For update_attributes, the new attributes for the object, such as the expiry date and activation date. A convenient way to provide this is via the vault_object_attrs helper function.
- ...: For update_attributes, additional key-specific properties to update. See keys.
- version: For set_version, the version ID or NULL for the current version.
- confirm: For delete, whether to ask for confirmation before deleting the key.

Details

The operations supported by a key will be those given by the key_ops argument when the key was created. By default, a newly created RSA key supports all the operations listed above: encrypt/decrypt, sign/verify, and wrap/unwrap. An EC key only supports the sign and verify operations.

A key can have multiple *versions*, which are automatically generated when a key is created with the same name as an existing key. By default, the most recent (current) version is used for key operations; use list_versions and set_version to change the version.

Value

For the key operations, a raw vector (for decrypt and unwrap, if as_raw=TRUE) or character vector.

For list_versions, a data frame containing details of each version.

For set_version, the key object with the updated version.

See Also

keys

Azure Key Vault documentation, Azure Key Vault API reference

Examples

```
## Not run:

vault <- key_vault("mykeyvault")

vault$keys$create("mynewkey")
# new version of an existing key
vault$keys$create("mynewkey", type="RSA", rsa_key_size=4096)

key <- vault$keys$get("mynewkey")
vers <- key$list_versions()
key$set_version(vers[2])

plaintext <- "some secret text"
```

```
ciphertext <- key$encrypt(plaintext)
decrypted <- key$decrypt(ciphertext, as_raw=FALSE)
decrypted == plaintext  # TRUE

dig <- openssl::sha2(charToRaw(plaintext))
sig <- key$sign(dig)
key$verify(sig, dig)  # TRUE

wraptext <- key$wrap(plaintext)
unwrap_text <- key$unwrap(wraptext, as_raw=FALSE)
plaintext == unwrap_text  # TRUE


## End(Not run)
```

---

keys *Encryption keys in Key Vault*

Description

This class represents the collection of encryption keys stored in a vault. It provides methods for managing keys, including creating, importing and deleting keys, and doing backups and restores. For operations with a specific key, see key.

Methods

This class provides the following methods:

```
create(name, type=c("RSA", "EC"), hardware=FALSE,
       ec_curve=NULL, rsa_key_size=NULL, key_ops=NULL,
       attributes=vault_object_attrs(), ...)
import(name, key, hardware=FALSE,
       attributes=vault_object_attrs(), ...)
get(name)
delete(name, confirm=TRUE)
list(include_managed=FALSE)
backup(name)
restore(backup)
```

Arguments

- `name`: The name of the key.
- `type`: For `create`, the type of key to create: RSA or elliptic curve (EC). Note that for keys backing a certificate, only RSA is allowed.
- `hardware`: For `create`, Whether to use a hardware key or software key. The former requires a premium key vault.
- `ec_curve`: For an EC key, the type of elliptic curve.

- `rsa_key_size`: For an RSA key, the key size, either 2048, 3072 or 4096.
- `key_ops`: A character vector of operations that the key supports. The possible operations are "encrypt", "decrypt", "sign", "verify", "wrapkey" and "unwrapkey". See key for more information.
- `attributes`: Optional attributes for the key, such as the expiry date and activation date. A convenient way to provide this is via the vault_object_attrs helper function.
- `key`: For `import`, the key to import. This can be the name of a PEM file, a JSON web key (JWK) string, or a key object generated by the openssl package. See the examples below.
- `hardware`: For `import`, whether to import this key as a hardware key (HSM). Only supported for a premium key vault.
- `...`: For `create` and `import`, other named arguments which will be treated as tags.
- `include_managed`: For `list`, whether to include keys that were created by Key Vault to support a managed certificate.
- `backup`: For `restore`, a string representing the backup blob for a key.

Value

For `get`, `create` and `import`, an object of class `stored_key`, representing the key itself. This has methods for carrying out the operations given by the `key_ops` argument.

For `list`, a vector of key names.

For `backup`, a string representing the backup blob for a key. If the key has multiple versions, the blob will contain all versions.

See Also

key, vault_object_attrs

Azure Key Vault documentation, Azure Key Vault API reference

Examples

```
## Not run:

vault <- key_vault("mykeyvault")

vault$keys$create("mynewkey")
vault$keys$create("myRSAkey", type="RSA", rsa_key_size=4096)
vault$keys$create("myECkey", type="EC", ec_curve="P-384")

vault$keys$list()
vault$keys$get("mynewkey")

# specifying an expiry date
today <- Sys.date()
vault$keys$create("mynewkey", attributes=vault_object_attrs(expiry_date=today+365))

# setting management tags
vault$keys$create("mynewkey", tag1="a value", othertag="another value")
```

```
# importing a key from a PEM file
vault$keys$import("importedkey1", "myprivatekey.pem")

# importing a key generated by OpenSSL
vault$keys$import("importedkey2", openssl::rsa_keygen())

# importing a JWK (which is a JSON string)
key <- openssl::read_key("myprivatekey.pem")
jwk <- jose::write_jwk(key)
vault$keys$import("importedkey3", jwk)

# backup and restore a key
bak <- vault$keys$backup("mynewkey")
vault$keys$delete("mynewkey", confirm=FALSE)
vault$keys$restore(bak)


## End(Not run)
```

`key_vault` *Azure Key Vault client*

Description

Azure Key Vault client

Usage

```
key_vault(url, tenant = "common", app = .az_cli_app_id, ...,
  domain = "vault.azure.net", token = NULL)
```

Arguments

`url`	The location of the vault. This can be a full URL, or the vault name alone; in the latter case, the `domain` argument is appended to obtain the URL.
`tenant, app, ...`	Authentication arguments that will be passed to AzureAuth::get_azure_token. The default is to authenticate interactively.
`domain`	The domain of the vault; for the public Azure cloud, this is `vault.azure.net`. Also the resource for OAuth authentication.
`token`	An OAuth token obtained via AzureAuth::get_azure_token. If provided, this overrides the other authentication arguments.

Details

This function creates a new Key Vault client object. It includes the following component objects for working with data in the vault:

- `keys`: A sub-object for working with encryption keys stored in the vault. See keys.
- `secrets`: A sub-object for working with secrets stored in the vault. See secrets.
- `certificates`: A sub-object for working with certificates stored in the vault. See certificates.
- `storage`: A sub-object for working with storage accounts managed by the vault. See storage.

See Also

keys, secrets, certificates, storage

Azure Key Vault documentation, Azure Key Vault API reference

Examples

```
## Not run:

key_vault("mykeyvault")
key_vault("https://mykeyvault.vault.azure.net")

# authenticating as a service principal
key_vault("mykeyvault", tenant="myaadtenant", app="app_id", password="password")

# authenticating with an existing token
token <- AzureAuth::get_azure_token("https://vault.azure.net", "myaadtenant",
                                    app="app_id", password="password")
key_vault("mykeyvault", token=token)


## End(Not run)
```

`list_deleted_key_vaults` *List soft-deleted Key Vaults*

Description

Method for the AzureRMR::az_subscription class.

Usage

```
list_deleted_key_vaults()
```

Value

This method returns a data frame with the following columns:

- `name`: The name of the deleted key vault.
- `location`: The location (region) of the vault.
- `deletion_date`: When the vault was soft-deleted.
- `purge_date`: When the vault is scheduled to be purged (permanently deleted).
- `protected`: Whether the vault has purge protection enabled. If TRUE, manual attempts to purge it will fail.

See Also

create_key_vault, get_key_vault, delete_key_vault, purge_key_vault, az_key_vault,

Azure Key Vault documentation, Azure Key Vault API reference

`purge_key_vault` *Purge a deleted Azure Key Vault*

Description

Method for the AzureRMR::az_subscription and AzureRMR::az_resource_group classes.

Details

This method permanently deletes a soft-deleted key vault. Note that it will fail if the vault has purge protection enabled.

Usage

```
purge_key_vault(name, location, confirm=TRUE)
```

Arguments

- `name`,`location`: The name and location of the key vault.
- `confirm`: Whether to ask for confirmation before permanently deleting the vault.

Value

NULL on successful purging.

See Also

create_key_vault, get_key_vault, delete_key_vault, list_deleted_key_vaults, az_key_vault,

Azure Key Vault documentation, Azure Key Vault API reference

Examples

```
## Not run:

rg <- AzureRMR::get_azure_login()$
    get_subscription("subscription_id")$
    get_resource_group("rgname")

# assuming the vault has soft-delete enabled, and is in the same location as its RG
rg$delete_key_vault("mykeyvault")
rg$purge_key_vault("mykeyvault", rg$location)


## End(Not run)
```

---

secrets *Stored secrets in Key Vault*

Description

This class represents the collection of secrets stored in a vault. It provides methods for managing secrets, including creating, importing and deleting secrets, and doing backups and restores.

Methods

This class provides the following methods:

```
create(name, value, content_type=NULL, attributes=vault_object_attrs(), ...)
get(name)
delete(name, confirm=TRUE)
list(include_managed=FALSE)
backup(name)
restore(backup)
```

Arguments

- `name`: The name of the secret.
- `value`: For `create`, the secret to store. This should be a character string or a raw vector.
- `content_type`: For `create`, an optional content type of the secret, such as "application/octet-stream".
- `attributes`: Optional attributes for the secret, such as the expiry date and activation date. A convenient way to provide this is via the vault_object_attrs helper function.
- `...`: For `create`, other named arguments which will be treated as tags.
- `include_managed`: For `list`, whether to include secrets that were created by Key Vault to support a managed certificate.
- `backup`: For `restore`, a string representing the backup blob for a secret.

Value

For get, and create, an object of class stored_secret, representing the secret. The actual value of the secret is in the value field.

For list, a vector of secret names.

For backup, a string representing the backup blob for a secret. If the secret has multiple versions, the blob will contain all versions.

See Also

vault_object_attrs

Azure Key Vault documentation, Azure Key Vault API reference

Examples

```
## Not run:

vault <- key_vault("mykeyvault")

vault$secrets$create("mysecret", "secret string")

vault$secrets$list()

secret <- vault$secrets$get("mysecret")
secret$value  # 'secret string'

# specifying an expiry date
today <- Sys.date()
vault$secrets$create("mysecret", attributes=vault_object_attrs(expiry_date=today+365))

# setting management tags
vault$secrets$create("mysecret", tag1="a value", othertag="another value")


## End(Not run)
```

---

storage_account *Managed storage account*

Description

This class represents a storage account that Key Vault will manage access to. It provides methods for regenerating keys, and managing shared access signatures (SAS).

This class represents a secret stored in a vault.

Fields

This class provides the following fields:

- `id`: The internal vault ID of the storage account.
- `resourceId`: The Azure resource ID of the storage account.
- `activeKeyName`: The current active storage account key.
- `autoRegenerateKey`: Whether Key Vault will manage the storage account's key.
- `regenerationPeriod`: How often the account key is regenerated, in ISO 8601 format.

This class provides the following fields:

- `value`: The value of the secret.
- `id`: The ID of the secret.
- `kid`: If this secret backs a certificate, the ID of the corresponding key.
- `managed`: Whether this secret's lifetime is managed by Key Vault. TRUE if the secret backs a certificate.
- `contentType`: The content type of the secret.

Methods

This class provides the following methods:

```
regenerate_key(key_name)
create_sas_definition(sas_name, sas_template, validity_period, sas_type="account",
                      enabled=TRUE, recovery_level=NULL, ...)
delete_sas_definition(sas_name, confirm=TRUE)
get_sas_definition(sas_name)
list_sas_definitions()
show_sas(sas_name)

update_attributes(attributes=vault_object_attrs(), ...)
remove(confirm=TRUE)
```

This class provides the following methods:

```
update_attributes(attributes=vault_object_attrs(), ...)
list_versions()
set_version(version=NULL)
delete(confirm=TRUE)
```

Arguments

- `key_name`: For `regenerate_key`, the name of the access key to regenerate.
- `sas_name`: The name of a SAS definition.
- `sas_template`: A string giving the details of the SAS to create. See 'Details' below.
- `validity_period`: How long the SAS should be valid for.

- sas_type: The type of SAS to generate, either "account" or "service".
- enabled: Whether the SAS definition. is enabled.
- recovery_level: The recovery level of the SAS definition.
- ...: For create_sas_definition, other named arguments to use as tags for a SAS definition. For update_attributes, additional account-specific properties to update. See storage_accounts.
- attributes: For update_attributes, the new attributes for the object, such as the expiry date and activation date. A convenient way to provide this is via the vault_object_attrs helper function.
- confirm: For delete and delete_sas_definition, whether to ask for confirmation before deleting.

- attributes: For update_attributes, the new attributes for the object, such as the expiry date and activation date. A convenient way to provide this is via the vault_object_attrs helper function.
- ...: For update_attributes, additional secret-specific properties to update. See secrets.
- version: For set_version, the version ID or NULL for the current version.
- confirm: For delete, whether to ask for confirmation before deleting the secret.

Details

create_sas_definition creates a new SAS definition from a template. This can be created from the Azure Portal, via the Azure CLI, or in R via the AzureStor package (see examples). get_sas_definition returns a list representing the template definition; show_sas returns the actual SAS.

regenerate_key manually regenerates an access key. Note that if the vault is setup to regenerate keys automatically, you won't usually have to use this method.

Unlike the other objects stored in a key vault, storage accounts are not versioned.

A secret can have multiple *versions*, which are automatically generated when a secret is created with the same name as an existing secret. By default, the most recent (current) version is used for secret operations; use list_versions and set_version to change the version.

The value is stored as an object of S3 class "secret_value", which has a print method that hides the value to guard against shoulder-surfing. Note that this will not stop a determined attacker; as a general rule, you should minimise assigning secrets or passing them around your R environment. If you want the raw string value itself, eg when passing it to jsonlite::toJSON or other functions which do not accept arbitrary object classes as inputs, use unclass to strip the class attribute first.

Value

For create_sas_definition and get_sas_definition, a list representing the SAS definition. For list_sas_definitions, a list of such lists.

For show_sas, a string containing the SAS.

For list_versions, a data frame containing details of each version.

For set_version, the secret object with the updated version.

See Also

storage_accounts

Azure Key Vault documentation, Azure Key Vault API reference

secrets

Azure Key Vault documentation, Azure Key Vault API reference

Examples

```
## Not run:

vault <- key_vault("mykeyvault")

# get the storage account details
library(AzureStor)
res <- AzureRMR::get_azure_login()$
    get_subscription("sub_id")$
    get_resource_group("rgname")$
    get_storage_account("mystorageacct")

stor <- vault$storage$create("mystor", res, "key1")

# Creating a new SAS definition
today <- Sys.time()
sasdef <- res$get_account_sas(expiry=today + 7*24*60*60, services="b", permissions="rw")
stor$create_sas_definition("newsas", sasdef, validity_period="P15D")

stor$show_sas("newsas")


## End(Not run)
## Not run:

vault <- key_vault("mykeyvault")

vault$secrets$create("mynewsecret", "secret text")
# new version of an existing secret
vault$secrets$create("mynewsecret", "extra secret text"))

secret <- vault$secrets$get("mynewsecret")
vers <- secret$list_versions()
secret$set_version(vers[2])

# printing the value will not show the secret
secret$value  # "<hidden>"


## End(Not run)
```

`storage_accounts` *Storage accounts in Key Vault*

Description

This class represents the collection of storage accounts managed by a vault. It provides methods for adding and removing accounts, and doing backups and restores. For operations with a specific account, see storage.

Methods

This class provides the following methods:

```
add(name, storage_account, key_name, regen_key=TRUE, regen_period=30,
    attributes=vault_object_attrs(), ...)
get(name)
remove(name, confirm=TRUE)
list()
backup(name)
restore(backup)
```

Arguments

- `name`: A name by which to refer to the storage account.
- `storage_account`: The Azure resource ID of the account. This can also be an object of class `az_resource` or `az_storage`, as provided by the AzureRMR or AzureStor packages respectively; in this case, the resource ID is obtained from the object.
- `key_name`: The name of the storage access key that Key Vault will manage.
- `regen_key`: Whether to automatically regenerate the access key at periodic intervals.
- `regen_period`: How often to regenerate the access key. This can be a number, which will be interpreted as days; or as an ISO-8601 string denoting a duration, eg "P30D" (30 days).
- `attributes`: Optional attributes for the secret. A convenient way to provide this is via the vault_object_attrs helper function.
- `...`: For `create` and `import`, other named arguments which will be treated as tags.
- `confirm`: For `remove`, whether to ask for confirmation before removing the account.
- `backup`: For `restore`, a string representing the backup blob for a key.
- `email`: For `set_contacts`, the email addresses of the contacts.

Value

For `get` and `add`, an object of class `stored_account`, representing the storage account itself.

For `list`, a vector of account names.

For `backup`, a string representing the backup blob for a storage account. If the account has multiple versions, the blob will contain all versions.

See Also

storage_account, vault_object_attrs

Azure Key Vault documentation, Azure Key Vault API reference

Examples

```
## Not run:

vault <- key_vault("mykeyvault")

# get the storage account details
library(AzureStor)
stor <- AzureRMR::get_azure_login()$
    get_subscription("sub_id")$
    get_resource_group("rgname")$
    get_storage_account("mystorageacct")
vault$storage$create("mystor", stor, "key1")

vault$storage$list()
vault$storage$get("mystor")

# specifying a regeneration period of 6 months
vault$storage$create("mystor", regen_period="P6M")

# setting management tags
vault$storage$create("mystor", tag1="a value", othertag="another value")

# backup and restore an account
bak <- vault$storage$backup("mystor")
vault$storage$delete("mystor", confirm=FALSE)
vault$storage$restore(bak)


## End(Not run)
```

---

vault_access_policy *Specify a key vault access policy*

Description

Specify a key vault access policy

Usage

```
vault_access_policy(principal, tenant = NULL, key_permissions = "all",
  secret_permissions = "all", certificate_permissions = "all",
  storage_permissions = "all")
```

Arguments

`principal`	The user or service principal for this access policy. Can be a GUID, or a user, app or service principal object from the AzureGraph package.
`tenant`	The tenant of the principal.
`key_permissions`	The permissions to grant for working with keys.
`secret_permissions`	The permissions to grant for working with secrets.
`certificate_permissions`	The permissions to grant for working with certificates.
`storage_permissions`	The permissions to grant for working with storage accounts.

Details

Client access to a key vault is governed by its access policies, which are set on a per-principal basis. Each principal (user or service) can have different permissions granted, for keys, secrets, certificates, and storage accounts.

Here are the possible permissions. The permission "all" means to grant all permissions.

- Keys: "get", "list", "update", "create", "import", "delete", "recover", "backup", "restore", "decrypt", "encrypt", "unwrapkey", "wrapkey", "verify", "sign", "purge"
- Secrets: "get", "list", "set", "delete", "recover", "backup", "restore", "purge"
- Certificates: "get", "list", "update", "create", "import", "delete", "recover", "backup", "restore", "managecontacts", "manageissuers", "getissuers", "listissuers", "setissuers", "deleteissuers", "purge"
- Storage accounts: "get", "list", "update", "set", "delete", "recover", "backup", "restore", "regeneratekey", "getsas", "listsas", "setsas", "deletesas", "purge"

Value

An object of class `vault_access_policy`, suitable for creating a key vault resource.

See Also

create_key_vault, az_key_vault

Azure Key Vault documentation, Azure Key Vault API reference

Examples

```
## Not run:

# default is to grant full access
vault_access_policy("user_id")

# use AzureGraph to specify a user via their email address rather than a GUID
usr <- AzureGraph::get_graph_login()$get_user("username@aadtenant.com")
```

```
vault_access_policy(usr)

# grant a service principal read access to keys and secrets only
svc <- AzureGraph::get_graph_login()$
    get_service_principal(app_id="app_id")
vault_access_policy(svc,
    key_permissions=c("get", "list"),
    secret_permissions=c("get", "list"),
    certificate_permissions=NULL,
    storage_permissions=NULL)


## End(Not run)
```

Index