Plug-in SDK

Contributors

Ivan Slavov ,Sabo Rusev ,Vladimir Dimitrov

Overview

The plug-in SDK is a set of jars and maven archetypes, distributed with each vRO installation. The SDK provides a set of interfaces, for communicating with the platform and accessing certain services:

- **Configuration service** - a service for storing data structures as configuration elements. Commonly used for storing connection data such as hosts, credentials etc.
- **SSL service** - a service which provides read access to the TrustStore of the platform
- **Cafe gateway service** - a service which allows an easy access of the vRA cafe services
- **SSO rest client service** - a more generic service of the Cafe gateway service
- **Solution authentication service** - a service, which provides read access to the solution user token
- **Scripting object contributor service** - a service which enables plug-ins to register scripting objects at runtime
- **Cipher service** - a service for encrypting/decrypting strings

The plug-in SDK is served as a maven repository which each vRO installation. You can access the repository at https://{vro_host}:8281/vco-repo/

Configuration service

Since	5.5.1

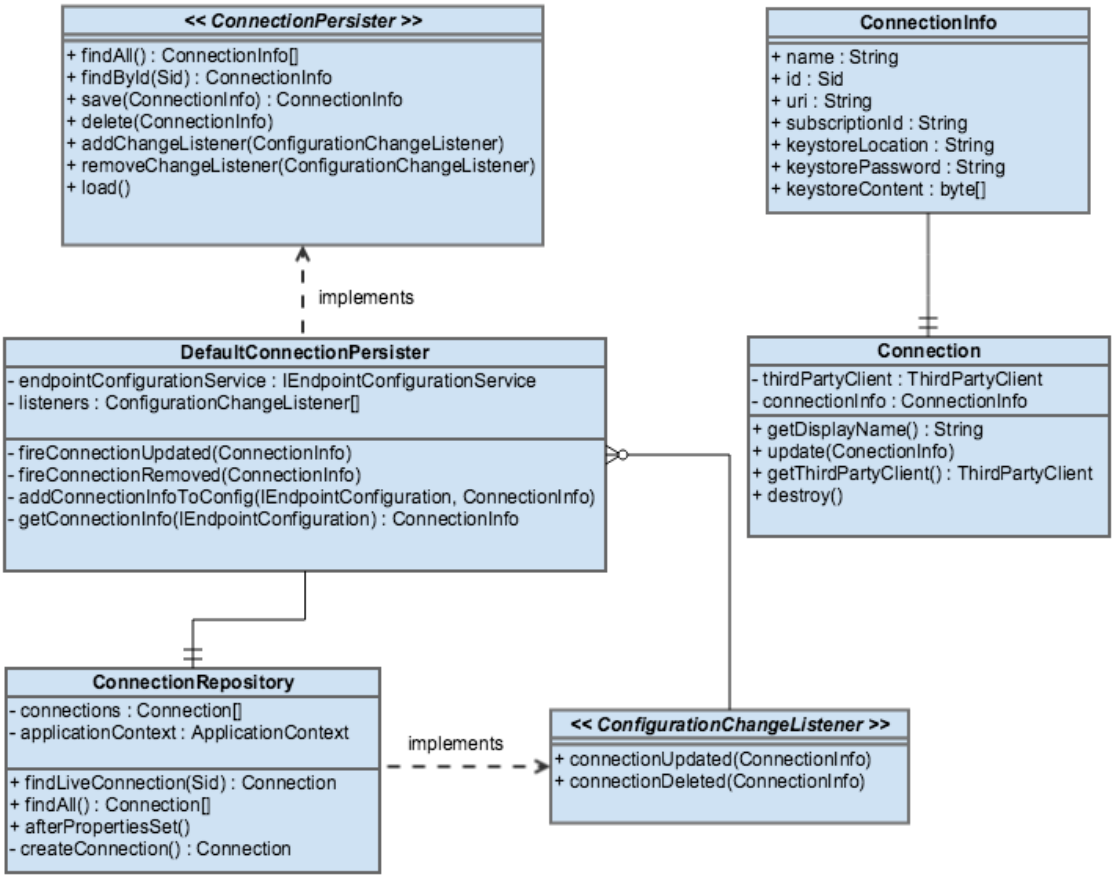
The configuration service is a service provided by the vRO platform, which allows storing endpoint data as a resource. The configuration data is represented as key-value pairs and currently supports the following value types OOTB: String, Password, Int, Long, Decimal, Boolean. The String value however can serve for multiple purposes. If you need to store binary data (for example a keystore) as part of a configuration, you can use the Apache commons codec library to base64 encode the data.

The configuration service consists of 2 interfaces which can be either injected (if the plug-in uses a spring context) or fetched from the service registry (if the plug-in does not use spring). In the latter case, two more interfaces are involved in order to get an instance of the configuration service.

Since	Interface	Description
5.5.1	IEndpointConfigurationService	A service, responsible for creating, reading, updating and storing configurations as vRO resources.
5.5.1	IEndpointConfiguration	Representation of a single configuration (stored as a vRO resource).
5.5.1	IServiceRegistryAdaptor	Entry point to the IServiceRegistry. This should be used in case the plug-in is not spring-based.
5.5.1	IServiceRegistry	A service locator for providing vRO services.

Sample code

The diagram below shows class diagram of a typical vRO configuration, implemented with the vRO Configuration Service:



Creating the connection info

The ConnectionInfo is a simple POJO which holds all the data needed to create/configure a connection to a third-party system. Note that this is not the actual connection (or Scripting object), but rather just the data which represents it.

ConnectionInfo Expand source

```
public class ConnectionInfo {

 /*
  * Name of the connection
  */
    private String name;

 /*
  * ID of the connection - can be String, UUID or whatever type you find suitable.
  */
    private final Sid id;

 /*
  * Service URI of the third party system
  */
    private String uri;

    private String subscriptionId;
    private String keystoreLocation;

 /*
  * Sensitive data - the keystore password
  */
    private String keystorePassword;

 /*
  * Some binary content - in this case we save a keystore
  */
    private byte[] keystoreContent;

 /*
  * Verify that each ConnectionInfo has an ID.
  */
    public ConnectionInfo() {
        this.id = Sid.unique();
    }

 /*
  * Verify that each ConnectionInfo has an ID.
  */
    public ConnectionInfo(Sid id) {
        super();
        this.id = id;
    }

 /*
  * Getters and setters
  */
    public String getUri() {
        return uri;
    }
    public void setUri(String uri) {
        this.uri = uri;
    }
    public String getSubscriptionId() {
        return subscriptionId;
    }
```

```
    public void setSubscriptionId(String subscriptionId) {
        this.subscriptionId = subscriptionId;
    }
    public String getKeystoreLocation() {
        return keystoreLocation;
    }
    public void setKeystoreLocation(String keystoreLocation) {
        this.keystoreLocation = keystoreLocation;
    }
    public String getKeystorePassword() {
        return keystorePassword;
    }
    public void setKeystorePassword(String keystorePassword) {
        this.keystorePassword = keystorePassword;
    }
    public Sid getId() {
        return id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public byte[] getKeystoreContent() {
        return keystoreContent;
    }
    public void setKeystoreContent(byte[] keystoreContent) {
        this.keystoreContent = keystoreContent;
    }

 /*
  * It is always a good idea to expose the fields of that object for logging and
debugging purposes.
  * Do not print the password.
  */
    @Override
    public String toString() {
        return "ConnectionInfo [name=" + name + ", id=" + id + ", uri=" + uri + ",
subscriptionId=" + subscriptionId
```

```
                + ", keystoreLocation=" + keystoreLocation + "]";
    }
}
```

Configuration change listener

The configuration change listener is an extension point of the configuration persister - the interface responsible for storing the data. It allows other components to subscribe to configuration events such as updating or deleting a ConnectionInfo. This is typical for a vRO plug-in, because it is usual to keep all the live connection/sessions in memory - a self managed cache. That way we can ensure that we have only one instance of a certain connection/session. However, it someone updates/deletes this connection using the ConnectionPersister, the cache where we store the live connection/session will be left in an inconsistent state. An implementation of the observer pattern is the resolution of this problem.

ConfigurationChangeListener Expand source

```
public interface ConfigurationChangeListener {
    /*
     * Invoked when the ConnectionInfo input is updated
     */

    void connectionUpdated(ConnectionInfo info);
    /*
     * Invoked when the ConnectionInfo input is deleted
     */
    void connectionRemoved(ConnectionInfo info);
}
```

Connection Persister

The ConnectionPersister is the class which is responsible for creating, reading, updating and deleting the ConnectionInfos. Roughly speaking, it is a wrapper of the IEndpointConfigurationService and hides the conversions between the IEndpointConfiguration and our ConnectionInfo.

Defining the interface

ConnectionPersister Expand source

```
public interface ConnectionPersister {
    /*
     * Returns a collection of all stored configurations (resources under
     * a folder with the plug-in name)
     */
    public List<ConnectionInfo> findAll();
    /*
     * Returns a collection by its ID or null if not found
     */
    public ConnectionInfo findById(Sid id);
    /*
     * Stores a connection info or updates it if already available.
     * The persister checks the availability of a connection by its ID
     */
    public ConnectionInfo save(ConnectionInfo connection);
    /*
     * Deletes a connection info. The persister will use the ID of the connection
     */
    public void delete(ConnectionInfo connectionInfo);
    /*
     * Allows us to subscribe to the events of the persister.
     * For example, if a connection is deleted, the persister will
     * trigger an event, notifying all subscribers.
     * This is an implementation of the observer pattern.
     */
    void addChangeListener(ConfigurationChangeListener listener);
    /*
     * Forces the persister to read all the configurations and trigger
     * the events. This method is invoked when the plug-in is loaded
     * on server start-up.
     */
    public void load();
}
```

Implementing the interface

DefaultConnectionPersister Expand source

```
@Component
public class DefaultConnectionPersister implements ConnectionPersister {
    private static final String CHARSET = "UTF-8";

 /*
  * A list of listeners, who have subscribed to any configuration events, such as
  * connection updates and deletions.
  */
    private final Collection<ConfigurationChangeListener> listeners;

 /*
  * Always use loggers
  */
```

```
    private static final Logger log =
LoggerFactory.getLogger(DefaultConnectionPersister.class);


 /*
  * Constants of the key names under which the connection values will be stored.
  */
    private static final String ID = "connectionId";
    private static final String NAME = "name";
    private static final String SUBSCRIPTION_ID = "sunscriptionId";
    private static final String KEYSTORE_CONTENT = "keystoreContent";
    private static final String KEYSTORE_PASSWORD = "keystorePassword";
    private static final String SERVICE_URI = "serviceUri";

 /*
  * The IEndpointConfigurationService will be injected through spring
  * if the plug-in has a spring context.
  */
    @Autowired
    private IEndpointConfigurationService endpointConfigurationService;

 /*
  * Persister constructor
  */
    public DefaultConnectionPersister() {
  //Initialise the listeners
        listeners = new CopyOnWriteArrayList<ConfigurationChangeListener>();
    }

 /*
  * Returns a collection of all stored configurations for this plug-in only
  * The service is aware of the plug-in name, thus will return only configurations for
this plug-in.
  */
    @Override
    public List<ConnectionInfo> findAll() {
        Collection<IEndpointConfiguration> configs;
        try {
   //Use the configuration service to retrieve all configurations.
   //The service is aware of the plug-in name, thus will return only configurations
for this plug-in.
            configs = endpointConfigurationService.getEndpointConfigurations();
            List<ConnectionInfo> result = new ArrayList<>(configs.size());

   //Iterate all the connections
            for (IEndpointConfiguration config : configs) {
    //Convert the IEndpointConfiguration to our domain object - the ConnectionInfo
                ConnectionInfo connectionInfo = getConnectionInfo(config);
                if (connectionInfo != null) {
                    log.debug("Adding connection info to result map: " +
connectionInfo);
                    result.add(connectionInfo);
                }
            }
            return result;
        } catch (IOException e) {
   log.debug("Error reading connections.", e);
            throw new RuntimeException(e);
        }
```

```
    }

 /*
  * Returns a ConnectionInfo by its ID
  * The service is aware of the plug-in name, thus cannot return a configuration for
another plug-in.
  */
    @Override
    public ConnectionInfo findById(Sid id) {
  //Sanity checks
        Validate.notNull(id, "Sid cannot be null.");

        IEndpointConfiguration endpointConfiguration;
        try {
   //Use the configuration service to retrieve the configuration service by its ID
            endpointConfiguration =
endpointConfigurationService.getEndpointConfiguration(id.toString());

   //Convert the IEndpointConfiguration to our domain object - the ConnectionInfo
            return getConnectionInfo(endpointConfiguration);
        } catch (IOException e) {
   log.debug("Error finding connection by id: " + id.toString(), e);
            throw new RuntimeException(e);
        }
    }

 /*
  * Save or update a connection info.
  * The service is aware of the plug-in name, thus cannot save the configuration
  * under the name of another plug-in.
  */
    @Override
    public ConnectionInfo save(ConnectionInfo connectionInfo) {
  //Sanity checks
        Validate.notNull(connectionInfo, "Connection info cannot be null.");
        Validate.notNull(connectionInfo.getId(), "Connection info must have an id.");

  //Additional validation - in this case we want the name of the connection to be
unique
  validateConnectionName(connectionInfo);
        try {
   //Find a connection with the provided ID. We don't expect to have an empty ID
            IEndpointConfiguration endpointConfiguration =
endpointConfigurationService
                    .getEndpointConfiguration(connectionInfo.getId().toString());
   //If the configuration is null, then we are performing a save operation
            if (endpointConfiguration == null) {
    //Use the configuration service to create a new (empty) IEndpointConfiguration.
    //In this case, we are responsible for assigning the ID of the configuration,
    //which is done in the constructor of the ConnectionInfo
                endpointConfiguration =
endpointConfigurationService.newEndpointConfiguration(connectionInfo.getId()
                        .toString());
            }

   //Convert the ConnectionInfo the IEndpointConfiguration
            addConnectionInfoToConfig(endpointConfiguration, connectionInfo);

   //Use the configuration service to save the endpoint configuration
```

```
endpointConfigurationService.saveEndpointConfiguration(endpointConfiguration);

   //Fire an event to all subscribers, that we have updated a configuration.
   //Pass the entire connectionInfo object and let the subscribers decide if they need
to do something
            fireConnectionUpdated(connectionInfo);
            return connectionInfo;
        } catch (IOException e) {
            log.error("Error saving connection " + connectionInfo, e);
            throw new RuntimeException(e);
        }
    }

 /*
  * Delete a connection info. The service is aware of the plug-in name, thus cannot
delete a configuration
  * from another plug-in.
  */
    @Override
    public void delete(ConnectionInfo connectionInfo) {
        try {
   //Use the configuration service to delete the connection info. The service uses the
ID

endpointConfigurationService.deleteEndpointConfiguration(connectionInfo.getId().toStri
ng());

   //Fire an event to all subscribers, that we have deleted a configuration.
   //Pass the entire connectionInfo object and let the subscribers decide if they need
to do something
            fireConnectionRemoved(connectionInfo);
        } catch (IOException e) {
            log.error("Error deleting endpoint configuration: " + connectionInfo, e);
            throw new RuntimeException(e);
        }
    }

 /*
  * This method is used to load the entire configuration set of the plug-in.
  * As a second step we fire a notification to all subscribers. This method
  * is used when the plug-in is being loaded (on server startup).
  */
    @Override
    public void load() {
        List<ConnectionInfo> findAll = findAll();
        for (ConnectionInfo connectionInfo : findAll) {
            fireConnectionUpdated(connectionInfo);
        }
    }

 /*
  * Attach a configuration listener.
  */
    @Override
    public void addChangeListener(ConfigurationChangeListener listener) {
        listeners.add(listener);
    }
```

```
 /*
  * A helper method which iterates all event subscribers and fires the
  * update notification for the provided connection info.
  */
    private void fireConnectionUpdated(ConnectionInfo connectionInfo) {
        for (ConfigurationChangeListener li : listeners) {
            li.connectionUpdated(connectionInfo);
        }
    }

 /*
  * A helper method which iterates all event subscribers and fires the
  * delete notification for the provided connection info.
  */
    private void fireConnectionRemoved(ConnectionInfo connectionInfo) {
        for (ConfigurationChangeListener li : listeners) {
            li.connectionRemoved(connectionInfo);
        }
    }

 /*
  * A helper method which converts our domain object the ConnectionInfo to an
IEndpointConfiguration
  */
    private void addConnectionInfoToConfig(IEndpointConfiguration config,
ConnectionInfo info) {
        try {
            config.setString(ID, info.getId().toString());
            config.setString(NAME, info.getName());
            config.setString(SUBSCRIPTION_ID, info.getSubscriptionId());
            config.setString(KEYSTORE_CONTENT, new String(info.getKeystoreContent(),
CHARSET));
            config.setPassword(KEYSTORE_PASSWORD, info.getKeystorePassword());
            config.setString(SERVICE_URI, info.getUri());
        } catch (UnsupportedEncodingException e) {
            log.error("Error converting ConnectionInfo to IEndpointConfiguration.",
e);
            throw new RuntimeException(e);
        }
    }

 /*
  * A helper method which converts the IEndpointConfiguration to our domain object the
ConnectionInfo
  */
    private ConnectionInfo getConnectionInfo(IEndpointConfiguration config) {
        ConnectionInfo info = null;
        try {
            Sid id = Sid.valueOf(config.getString(ID));
            info = new ConnectionInfo(id);
            info.setName(config.getString(NAME));
            info.setUri(config.getString(SERVICE_URI));
            info.setSubscriptionId(config.getString(SUBSCRIPTION_ID));
            info.setKeystorePassword(config.getPassword(KEYSTORE_PASSWORD));

info.setKeystoreContent(config.getString(KEYSTORE_CONTENT).getBytes(CHARSET));
        } catch (IllegalArgumentException | UnsupportedEncodingException e) {
            log.warn("Cannot convert IEndpointConfiguration to ConnectionInfo: " +
config.getId(), e);
```

```
        }
        return info;
    }
    private void validateConnectionName(ConnectionInfo connectionInfo) {
        ConnectionInfo configurationByName =
getConfigurationByName(connectionInfo.getName());
        if (configurationByName != null
                &&
!configurationByName.getId().toString().equals(connectionInfo.getId().toString())) {
            throw new RuntimeException("Connection with the same name already exists:
" + connectionInfo);
        }
    }
    private ConnectionInfo getConfigurationByName(String name) {
        Validate.notNull(name, "Connection name cannot be null.");
        Collection<ConnectionInfo> findAllClientInfos = findAll();
        for (ConnectionInfo info : findAllClientInfos) {
            if (name.equals(info.getName())) {
                return info;
            }
        }
```

```
        return null;
    }
}
```

Connection repository

The ConnectionRepository is a plug-in's local cache - the place where we keep all live connections. Any read operation related to a live connection must be performed through that interface. The difference between the ConnectionPersister and the ConnectionRepository is that the persister only manages ConnectionInfos - the POJOs which stay behind the live connections, whereas the ConnectionRepository is responsible for providing one and the same instance representing a certain third-party connection. In other words - the ConnectionRepository deals with Connections, and the ConnectionPersister deals with ConnectionInfos.

The repository only provides two methods - findLiveConnection(..) and findAll(..). If you're wondering how to edit (or delete) a certain connection from the repository (which is a local cache) - you should use the ConnectionPersister. Since the ConnectionRepository is subscribed to update/delete events of the ConnectionPersister, once a connection is updated/deleted from the persister, the change will be propagated to the repository.

ConnectionRepository Expand source

```
/*
 * The ConnectionRepository implements the ConfigurationChangeListener, because we
want to be subscribed to all
 * changes related to configurations. ApplicationContextAware and InitializingBean are
spring-related interfaces.
 */
@Component
public class ConnectionRepository implements ApplicationContextAware,
InitializingBean, ConfigurationChangeListener {


 /*
  * Injecting the ConnectionPersister
  */
    @Autowired
    private ConnectionPersister persister;

    private ApplicationContext context;

 /*
  * The local map (cache) of live connections
  */
    private final Map<Sid, Connection> connections;

    public ConnectionRepository() {
        connections = new ConcurrentHashMap<Sid, Connection>();
    }

 /*
  * The public interface returns a live connection by its ID
  */
    public Connection findLiveConnection(Sid anyId) {
        return connections.get(anyId.getId());
    }

 /*
  * The public interface returns all live connections from the local cache
  */
    public Collection<Connection> findAll() {
```

```
        return connections.values();
    }

 /*
  * Spring-specifics - storing a reference to the spring context
  */
    @Override
    public void setApplicationContext(ApplicationContext context) throws
BeansException {
        this.context = context;
    }

 /*
  * Spring specifics - this method is being called automatically by the spring
container
  * after all the fields are set and before the bean is being provided for usage.
  * This method will be called when the plug-in is being loaded - on server start-up.
  */
    @Override
    public void afterPropertiesSet() throws Exception {
  //Subscribing the Repository for any configuration changes that occur in the
Persister
        persister.addChangeListener(this);

  //Initialising the Persister. By doing that, the persister will invoke the
connectionUpdated() method
  //and since we are subscribed to those events, the local cache will be populated
with all the available connections.
        persister.load();
    }

    private Connection createConnection(ConnectionInfo info) {
  //This call will create a new spring-managed bean from the context
        return (Connection) context.getBean("connection", info);
    }

 /*
  * This method will be called from the ConnectionPersister when a new connection
  * is added or an existing one is updated.
  */
    @Override
    public void connectionUpdated(ConnectionInfo info) {
        Connection live = connections.get(info.getId());
        if (live != null) {
            live.update(info);
        } else {
            // connection just added, create it
            live = createConnection(info);
            connections.put(info.getId(), live);
        }
    }

 /*
  * This method will be called from the ConnectionPersister when a connection
  * is removed.
  */
    @Override
    public void connectionRemoved(ConnectionInfo info) {
        Connection live = connections.remove(info.getId());
```

```
        if (live != null) {
            live.destroy();
```

```
        }
    }
}
```

Connection

The representation of a live connection. There must be one and only one instance of that object per connection. Those instances are kept in the ConnectionRepository.

Connection Expand source

```
@Component
@Qualifier(value = "connection")
@Scope(value = "prototype")
public class Connection {
    /*
     * Some third party client which enables the communications between
     * vRO and the third party system. Can be a http client, SSH client etc.
     */
    private ThirdPartyClient thirdPartyClient;
    /*
     * The connectionInfo which stands behind this live connection.
     */
    private ConnectionInfo connectionInfo;
    /*
     * There is no default constructor, the Connection must be
     * initialised only with a connection info argument.
     */
    public Connection(ConnectionInfo info) {
        init(info);
    }
    public synchronized ConnectionInfo getConnectionInfo() {
        return connectionInfo;
    }
    public String getDisplayName() {
        return getConnectionInfo().getName() + " [" +
getConnectionInfo().getSubscriptionId() + "]";
    }
    /*
     * Updates this connection with the provided info. This operation will
     * destroy the existing third party client, causing all associated operations to
fail.
     */
    public synchronized void update(ConnectionInfo connectionInfo) {
        if (this.connectionInfo != null &&
!connectionInfo.getId().equals(this.connectionInfo.getId())) {
            throw new IllegalArgumentException("Cannot update using different id");
        }
        destroy();
        init(connectionInfo);
    }
    private void init(ConnectionInfo connectionInfo) {
        this.connectionInfo = connectionInfo;
    }
    /*
     * Lazy-initialises the ThirdPartyClient instance
```

```
 */
public synchronized ThirdPartyClient getThirdPartyClient() {
    if (thirdPartyClient == null) {
        thirdPartyClient = new ThirdPartyClient();
        //Use the connectionInfo to setup the ThirdPartyClient properties
        //thirdPartyClient.set(..);
    }
    return thirdPartyClient;
}
public synchronized void destroy() {
    //Destroy the client, releasing all current connections and
    //possibly cleaning the resources.
    thirdPartyClient.close();
    //Set the client to null so that it can be reinitialized by the
    //getThirdPartyClient() method
    thirdPartyClient = null;
}
```

```
}
```

Cluster awareness of configurations

Because vRO instances are able to work in a cluster, a question arises about transferring configuration data . When a node changes configuration data (add inventory objects etc.), these should be made available to the workflow engine of other nodes.

This is not happening automatically and plugin's developer should write code which :

1. detects that this change is happening. The instrument for this is IEndpointConfigurationService.getVersion() method, which returns the current version of the configuration data in the database. The code can detect the change by comparing this version to stored one, e.g.

```
public class DefaultConnectionPersister implements ConnectionPersister {

 @Autowired
    private IEndpointConfigurationService cachingEndpointConfigurationService;

 private String configurationVersion;

 public boolean changed() throws IOException {
  return configurationVersion == null || configurationVersion.isEmpty() ||
!configurationVersion.equals(cachingEndpointConfigurationService.getVersion());
 }

}
```

   This check should be called on every access to objects depending on configuration data (inventory objects etc.)

   If change is detected we will load configuration data (Step 2)

   Note that IEndpointConfigurationService.getVersion() is cached on the platform side.
   a. When working with platform version 7.0.0 and above, the caching time is maximum 2 heart-beat intervals, i.e. if heart-beat interval is 5 seconds and node 1 changes configuration data , getVersion() call on other nodes will return cached value for maximum 10 sconds
   b. When working with platform versions below 7.0.0, this version is not cached. But caching is important , because now we will have to check getVersion() much more frequently. For that reason , such a caching is provided additionally (see below) and TTL period in this case is 10 seconds

   To ensure that IEndpointConfigurationService.getVersion() is working properly on every platform, the developer must use the proper implementation of IEndpointConfigurationService. If your plugin will work only with platform version 7.0.0 and above, you can skip this section
   a. Make sure you should depend on platform version 7.0.0 and above
   b. Add compile-time dependency to o11n-plugin-tools artifact in maven pom.xml, so this artifact will be packaged in the plugin's archive

```
<dependency>
   <groupId>com.vmware.o11n</groupId>
   <artifactId>o11n-plugin-tools</artifactId>
   <version>${vco.version}</version>
</dependency>
```

   c. Provide proper implementation of IEndpointConfigurationService
      i. For spring-aware plugins, the proper component's name is **cachingEndpointConfigurationService.** To use it **:**
         1. If using spring , this should be your property name.

```
@Autowired
 private IEndpointConfigurationService
cachingEndpointConfigurationService;
```

2. Add in your plugin's application context xml:

```
<context:component-scan
base-package="com.vmware.o11n.plugin.sdk.endpoints.extensions"
annotation-config="true"/>
```

ii. For plugins which are not spring-aware, just look for the proper version up, for example in setServiceRegistry method

```
endpointConfigurationService =
EndpointConfigurationServiceFactory.lookupEndpointConfigurationService
(registry);
```

2. reloads configuration data - you can fetch all the data through the endpointConfiguratonService, determine which entries have been changed from other nodes and proceed to step 3
3. updates state of inventory objects, if needed, based on the configuration data you have loaded

Reading the configuration files of other plug-ins

Since	Interface	Description
5.5.1	IEndpointConfigurationService	A service, responsible for creating, reading, updating and storing configurations as vRO resources.
5.5.1	IEndpointConfiguration	Representation of a single configuration (stored as a vRO resource).

Some plug-ins rely on the configuration files of other plug-ins. The IEndpointConfigurationService provides a method which enables you to get a read-only configuration of another plug-in.

Sample code

```
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import ch.dunes.vso.sdk.endpoints.IEndpointConfiguration;
import ch.dunes.vso.sdk.endpoints.IEndpointConfigurationService;

//Reader for vCenter configurations
public class VcenterConnectionReader {

    @Autowired
    private IEndpointConfigurationService endpointConfigurationService;


    public List<VcHost> loadVcPluginHosts() throws IOException {
        List<VcHost> hosts = new ArrayList<VcHost>();
        try {
   //Read the configuration of another plug-in by its name
```

```
endpointConfigurationService.getEndpointConfigurationServiceForPlugin("VC");

   //Iterate through the IEndpointConfiguration objects and construct your own model
objects
            for (IEndpointConfiguration configuration :
endpointConfigurationService.getEndpointConfigurations()) {
                String passwordEncrypted =
configuration.getPassword("administratorPassword");
                Boolean isSharedLoginMode =
configuration.getAsBoolean("sharedLoginMode");
                String administratorUsername =
configuration.getString("administratorUsername");
                Boolean isEnabled = configuration.getAsBoolean("enabled");
                String url = configuration.getString("url");
                VcHost vcHost = new VcHost();
                vcHost.setActive(isEnabled);
                vcHost.setPassword(passwordEncrypted);
                vcHost.setUsername(administratorUsername);
                vcHost.setSharedLoginMode(isSharedLoginMode);
                vcHost.setVcUri(url);
                if (isEnabled) {
                    hosts.add(vcHost);
                }
            }
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
        return hosts;
    }

 //A model class representing vCenter connections
    public class VcHost {
        private Boolean isEnabled;
        private String url;
        private Boolean isSharedLoginMode;
        private String administratorUsername;
        private String passwordEncrypted;
        public Boolean getIsEnabled() {
            return isEnabled;
        }
        public void setIsEnabled(Boolean isEnabled) {
            this.isEnabled = isEnabled;
        }
        public String getUrl() {
            return url;
        }
        public void setUrl(String url) {
            this.url = url;
        }
        public Boolean getIsSharedLoginMode() {
            return isSharedLoginMode;
        }
        public void setIsSharedLoginMode(Boolean isSharedLoginMode) {
            this.isSharedLoginMode = isSharedLoginMode;
        }
        public String getAdministratorUsername() {
            return administratorUsername;
        }
```

```
public void setAdministratorUsername(String administratorUsername) {
    this.administratorUsername = administratorUsername;
}
public String getPasswordEncrypted() {
    return passwordEncrypted;
}
public void setPasswordEncrypted(String passwordEncrypted) {
    this.passwordEncrypted = passwordEncrypted;
}
public void setActive(Boolean isEnabled) {
    this.isEnabled = isEnabled;
}
public void setVcUri(String url) {
    this.url = url;
}
public void setSharedLoginMode(Boolean isSharedLoginMode) {
    this.isSharedLoginMode = isSharedLoginMode;
}
public void setUsername(String administratorUsername) {
    this.administratorUsername = administratorUsername;
}
public void setPassword(String passwordEncrypted) {
    this.passwordEncrypted = passwordEncrypted;
```

```
        }
    }
}
```

SSL service

Since	5.5.1

The SSL service is a service provided by the vRO platform, which provides read-only access to the vCO servers trustore.

Writing to the truststore is achieved with the following workflows, which are part of the platform:

Workflow	Description
Library / Configuration / SSL Trust Manager / Import a certificate from URL	Imports a trusted certificate from a remote URL. The certificate will be imported with an auto-generated alias.
Library / Configuration / SSL Trust Manager / Import a certificate from URL using proxy server	Imports a trusted certificate from a remote URL through proxy. The certificate will be imported with an auto-generated alias.
Library / Configuration / SSL Trust Manager / Import a certificate from URL with certificate alias	Imports a trusted certificate from a remote URL. The certificate will be imported with a provided alias name.
Library / Configuration / SSL Trust Manager / Import a trusted certificate from a file	Imports a trusted certificate from a file. The file must be in der format.

The provided workflows a meant to be used by the configuration workflows of plug-ins, which require to establish SSL/TLS connections.

Sample code

Here is a sample code how we can call the getService() method from IServiceRegistry and get an ISslService in our plugin.
(If the plugin supports spring we can autowire the sslService)

Register sslService Expand source

```
import ch.dunes.vso.sdk.IServiceRegistryAdaptor;
import ch.dunes.vso.sdk.ssl.ISslService;

public final class PluginAdaptor implements IServiceRegistryAdaptor {
 @Override
    public void setServiceRegistry(IServiceRegistry registry) {
        ISslService sslService = (ISslService)
registry.getService(IServiceRegistry.SSL_SERVICE);
    }
}
```

Once we have the sslService we can persist it in our plugin and then use it when needed. It provides two things - a SslContext and HostNameVerifier

ISslService Expand source

```
public interface ISslService {
    SSLContext newSslContext(String protocol) throws NoSuchAlgorithmException;

    HostnameVerifier newHostNameVerifier();
}
```

After we have our sslService, we can create a new SslContext and specify the desired protocol like so:

SSLContext Expand source

```
SSLContext sslContext = sslService.getSslContext("SSL");
```

And after we have a sslContext we can create a SSLSocketFactory:

SSLSocketFactory Expand source

```
SSLSocketFactory factory = sslContext.getSocketFactory();
```

This factory can be used for creating socket with specifing (optional) - host, port, autoClose, localAddress, localPort, localHost.

Now when we have a SSLSocketFactory we can set it to a HttpsUrlConnection:

Set SslSocketFactory to a https connection Expand source

```
URL url = new URL(serverUrl); //a string with some serverUrl
URLConnection connection = url.openConnection();
HttpsURLConnection httpsConnection = (HttpsURLConnection) connection;
SSLContext sslContext = sslService.newSslContext("SSL");
httpsConnection.setSSLSocketFactory(sslContext.getSocketFactory());
httpsConnection.setHostnameVerifier(sslService.newHostNameVerifier());
```

The HostNameVerifier is used when an URLConnection is opened like so:

HostnameVerifier Expand source

```
URL url = new URL(serverUrl); //a string with some serverUrl
URLConnection connection = url.openConnection();
HostnameVerifier hostnameVerifier = sslService.newHostNameVerifier();
((HttpsURLConnection) connection).setHostnameVerifier(hostnameVerifier);
```

Cafe gateway service

Since	5.5.2

The Cafe gateway service is a service provided by the vRO platform which exposes the RestClient of the CAFE endpoints. This is a more limited version of the SSO rest client service. It creates a preconfigured RestClient which communicates with the vRealize Automation cafe services.

Sample code

The code example below shows how to get the service if the plug-in is spring enabled.

Accessing the service - spring plug-ins Expand source

```
//Object must be declared in the spring context
public class MyDomainObject {

    @Autowired
    private CafeGateway cafeGateway;

}
```

The code example below shows how to get the service if the plug-in is not spring enabled.

Accessing the service - non-spring plug-ins Expand source

```
import ch.dunes.vso.sdk.cafe.CafeGateway;

public class PluginAdaptor implements IServiceRegistryAdaptor {

 private IServiceRegistry registry;
    private CafeGateway cafeGateway;

 @Override
    public void setServiceRegistry(IServiceRegistry registry) {
  this.registry = registry;
    }

 public CafeGateway getCafeGateway() {
  if (cafeGateway == null) {
   cafeGateway = (CafeGateway)
registry.getService(IServiceRegistry.CAFE_GATEWAY_SERVICE);
  }
  return cafeGateway;
 }

}
```

The code below shows an example of how to read log events from vRA

Building a ScriptingObjectDefinition Expand source

```
import org.springframework.data.domain.PageRequest;

import org.springframework.data.domain.Pageable;
import com.vmware.vcac.core.eventlog.rest.client.service.EventLogService;
import com.vmware.vcac.eventlog.rest.stubs.Event;
import com.vmware.vcac.platform.data.services.RegistryCatalog;
import com.vmware.vcac.platform.rest.client.RestClient;
import com.vmware.vcac.platform.rest.data.PagedResources;

//Provided the service is already instantiated
CafeGateway cafeGateway;

//Retrieve a rest client from the gateway service
RestClient restClient = cafeGateway.restClientForSolutionUserByServiceAndEndpointType(
    RegistryCatalog.EVENTLOG_SERVICE.getServiceTypeId(),
    RegistryCatalog.EVENTLOG_SERVICE.getDefaultEndPointTypeId());


//Instantiate the service object with the newly created rest client
EventLogService service = new EventLogService(restClient);

//Create a pageable request
Pageable pageable = new PageRequest(1, 15);

//Request the events
PagedResources<Event> page = service.getAllEvents(pageable);


//Get the content of the page
Collection<Event> events = page.getContent();
```

SSO rest client service

Since	6.0.1

The SSO rest client service is a service provided by the vRO platform which exposes the RestClient of the CAFE endpoints. This is a more extended version of the Cafe gateway service, because it allows you to specify the host and root path of the rest client, as opposed to the Cafe gateway service which only communicates with the cafe server, being configured as authentication provider.

Sample code

The code example below shows how to get the service if the plug-in is spring enabled.

Accessing the service - spring plug-ins

Expand source

```
//Object must be declared in the spring context
public class MyDomainObject {

    @Autowired
    private SsoRestClientFactory restClientFactory;

}
```

The code example below shows how to get the service if the plug-in is not spring enabled.

Accessing the service - non-spring plug-ins

Expand source

```
public class PluginAdaptor implements IServiceRegistryAdaptor {

 private IServiceRegistry registry;
    private SsoRestClientFactory ssoRestClientFactory;

 @Override
    public void setServiceRegistry(IServiceRegistry registry) {
  this.registry = registry;
    }

 public SsoRestClientFactory getSsoRestClientService() {
  if (ssoRestClientFactory == null) {
   this.ssoRestClientFactory = (SsoRestClientFactory)
registry.getService(IServiceRegistry.SSO_REST_CLIENT_SERVICE));
  }
  return ssoRestClientFactory;
 }
}
```

createSamlAuthSession Expand source

```
import com.vmware.o11n.sdk.rest.client.authentication.Authentication;
import com.vmware.o11n.sdk.rest.client.VcoSessionOverRestClient;
import com.vmware.o11n.sdk.rest.client.configuration.ConnectionConfiguration;
import com.vmware.o11n.plugin.vcoconn.config.ServerConfigurationManager;
import ch.dunes.vso.sdk.cafe.SsoRestClientFactory;
import ch.dunes.vso.sdk.IServiceRegistry;
import ch.dunes.vso.sdk.api.ISamlToken;

public class SampleSessionFactory implements VcoSessionFactory {
 private IServiceRegistry registry;
 private SsoRestClientFactory ssoRestClientFactory;

 public SampleSessionFactory (URI uri, ConnectionConfiguration
connectionConfiguration) {
     IServiceRegistry registry =
ServerConfigurationManager.getInstance().getServiceRegistry();
        this.ssoRestClientFactory = ((SsoRestClientFactory) registry
                .getService(IServiceRegistry.SSO_REST_CLIENT_SERVICE));
 }

 VcoSession createSamlAuthSession(Authentication auth, ConnectionConfiguration config)
{
      ISamlToken token = auth.getToken();
       RestClient client =
ssoRestClientFactory.credentialPropagatingRestClientByUri(uri, token, config);
       return new VcoSessionOverRestClient(client);
 }
}
```

Solution authentication service

Since	7.0.0

The Solution Authentication Service is a service provided by the vRO platform which provides access to the solution user token. The service does not provide direct access to the token, but rather creates an authentication object which can be used when constructing a Rest client to consume the vRA APIs.

This service is available only when vRO is configured to work with vRA authentication and is usually used when you need to issue RESTful calls against a vRA service.

Sample code

The code example below shows how to get the service if the plug-in is spring enabled.

Accessing the service - spring plug-ins Expand source

```
//Object must be declared in the spring context
public class MyDomainObject {

    @Autowired
    private SolutionAuthenticationService solutionAuthenticationService;

}
```

The code example below shows how to get the service if the plug-in is not spring enabled.

Accessing the service - non-spring plug-ins Expand source

```
public class PluginAdaptor implements IServiceRegistryAdaptor {


 private IServiceRegistry registry;
    private SolutionAuthenticationService solutionAuthenticationService;

 @Override
    public void setServiceRegistry(IServiceRegistry registry) {
  this.registry = registry;
    }

 public SolutionAuthenticationService getSolutionAuthenticationService() {
  if (solutionAuthenticationService == null) {
   solutionAuthenticationService = (SolutionAuthenticationService)
registry.getService(IServiceRegistry.SOLUTION_AUTHENTICATION_SERVICE);
  }
  return solutionAuthenticationService;
 }

}
```

Scripting object contributor service

Since	7.0.0

The scripting object contributor service is a service provided by the vRO platform, which enables plug-ins to register scripting objects at runtime, without having to restart the server.

Since	Interface	Description
7.0.0	ScriptingObjectsContributor	Service endpoint to contribute scripting objects at runtime
7.0.0	ScriptingObjectDefinition	Definition of a scripting object
7.0.0	ScriptingAttributeDefinition	Definition of an attribute of a scripting object
7.0.0	ScriptingMethodDefinition	Definition of a method of a scripting object
7.0.0	ScriptingConstructorDefinition	Definition of a constructor of a scripting object

Sample code

The code example below shows how to get the service if the plug-in is spring enabled.

Accessing the service - spring plug-ins Expand source

```
//Object must be declared in the spring context
public class MyDomainObject {

    @Autowired
    private ScriptingObjectsContributor scriptingObjectContributor;

}
```

The code example below shows how to get the service if the plug-in is not spring enabled.

Accessing the service - non-spring plug-ins Expand source

```
public class PluginAdaptor implements IServiceRegistryAdaptor {


 private IServiceRegistry registry;
    private ScriptingObjectsContributor scriptingObjectContributor;

 @Override
    public void setServiceRegistry(IServiceRegistry registry) {
  this.registry = registry;
    }

 public ScriptingObjectsContributor getScriptingObjectContributor() {
  if (scriptingObjectContributor == null) {
   scriptingObjectContributor = (ScriptingObjectsContributor)
registry.getService(IServiceRegistry.SCRIPTING_OBJECTS_CONTRIBUTOR_SERVICE);
  }
  return scriptingObjectContributor;
 }

}
```

The code below shows how to contribute a scripting object which has a single attribute.

Building a ScriptingObjectDefinition Expand source

```
//Provided the service is already instantiated
ScriptingObjectsContributor scriptingObjectContributor;

//Init a list of attributes of the scripting object
List<ScriptingAttributeDefinition> attributes = new LinkedList<>();

//Create a single scripting attribute of type string
ScriptingAttributeDefinition attribute = ScriptingAttributeDefinition.newBuilder()
    .scriptName("scriptingAttributeName") //The attribute will appear with this name
in the orchestrator
    .javaName("scriptingAttributeName") //The attribute name in the corresponding Java
class
    .type("string") //The attribute type
    .description("Description of the attribute") //Description of the attribute
    .build();
attributes.add(attribute);

//Create a scripting object, setting the above created attribute
ScriptingObjectDefinition object = ScriptingObjectDefinition.newBuilder()
    .scriptingName("scriptingObjectName") //The name of the scripting object as it
will appear in the orchestrator
    .javaClassName(MyClass.class.getName()) //The corresponding java class
    .attributes(attributes) //The above created attributes
    .description(enumeration.getDocumentation()) //Description of the scripting object
    .dynamic(true)
    .dynamicInvocation(true)
    .build();

//Invoke the service
scriptingObjectContributor.contribute(Collections.singletonList(object));
```

Cipher service

Since	7.0.0

The cipher service is a service provided by the vRO platform, which uses vRO's encryption algorithm to encrypt/decrypt strings. This service should be used when a plug-in needs to store sensitive data.

Sample code

The code example below shows how to get the service if the plug-in is spring enabled.

Accessing the service - spring plug-ins Expand source

```
//Object must be declared in the spring context
public class MyDomainObject {

    @Autowired
    private ICipher cipherService;
}
```

The code example below shows how to get the service if the plug-in is not spring enabled.

Accessing the service - non-spring plug-ins Expand source

```
public class PluginAdaptor implements IServiceRegistryAdaptor {

 private IServiceRegistry registry;
    private ICipher cipherService;

 @Override
    public void setServiceRegistry(IServiceRegistry registry) {
  this.registry = registry;
    }

 public ICipher getCipherService() {
  if (cipherService == null) {
   cipherService = registry.getService(IServiceRegistry.CIPHER_SERVICE);
  }
  return cipherService;
 }
}
```

Once we have the cipher we can then call the encrypt and decrypt methods:

Accessing the service - non-spring plug-ins Expand source

```
public class MyCustomModelClass {
    private ICipher cipherService;

   //Encrypts a string
    cipherService.encrypt("text-to-encrypt");
    //Decrypt a vRO encrypted string
    cipherService.decrypt("..."); // the string for decryption
}
```

If the decryption fails it will failback to the legacy decryption algorithm used in older versions.