Plug-In SDK Guide for vRealize Orchestrator

Development Guide
# Table of Contents

Overview .......................................................................................................................... 3

Configuration Service ...................................................................................................... 3
  Creating ConnectionInfo ................................................................................................. 4
  Creating the Configuration Change Listener ................................................................. 6
  Connection Persister ....................................................................................................... 7
  Connection Repository .................................................................................................... 12
  Connection ..................................................................................................................... 14
  Implement a Cluster Awareness of Configurations ....................................................... 16
  Reading Configuration Files of Other Plug-Ins ............................................................. 17

SSL Service ....................................................................................................................... 19
  Retrieve sslService from Service Registry ................................................................. 20

CAFÉ Gateway Service .................................................................................................... 21

SSO REST Client Service ................................................................................................. 22

Solution Authentication Service ....................................................................................... 23

Scripting Object Contributor Service ............................................................................... 24

Cipher Service .................................................................................................................. 26
Overview

The Software Development Kit for Orchestrator plug-ins is a set of libraries and Maven archetypes that are distributed with every installation of vRealize Orchestrator. The Plug-in SDK provides a collection of interfaces for communicating with the Orchestrator platform and accessing different services.

<table>
<thead>
<tr>
<th>SERVICE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuration service</td>
<td>Stores data as configuration elements, such as host names, credentials, and others.</td>
</tr>
<tr>
<td>SSL service</td>
<td>Provides read access to the Trust store of the platform.</td>
</tr>
<tr>
<td>CAFÉ gateway service</td>
<td>Provides access to the CAFÉ services in vRealize Automation.</td>
</tr>
<tr>
<td>SSO REST client service</td>
<td>A service that is more generic than the CAFÉ gateway service.</td>
</tr>
<tr>
<td>Solution authentication service</td>
<td>Provides read access to the solution user token.</td>
</tr>
<tr>
<td>Scripting object contributor service</td>
<td>Makes possible for a plug-in to register scripting objects at runtime.</td>
</tr>
<tr>
<td>Cipher service</td>
<td>Encrypts and decrypts strings.</td>
</tr>
</tbody>
</table>

The Plug-in SDK is available as a Maven repository as part of the Orchestrator Appliance. You can access the repository at https://orchestrator_server_IP_or_DNS_name:8281/vco-repo/.

Configuration Service

Available in version 5.5.1 and later.

The Configuration service is included in the Orchestrator platform with version 5.5.1. This service makes it possible to store data as a resource. The configuration data consists of key-value pairs and supports String, Password, Integer, Long, Decimal, and Boolean data types.

The String value serves multiple purposes. For example, if you want to store binary data, such as a keystore, as part of a configuration, you can encode the data in Base64.

The Configuration service includes two interfaces. If the plug-in uses a Spring context, the interfaces are injected. If the plug-in does not use Spring, the interfaces are retrieved from the service registry. In this case, you need two more interfaces to retrieve an instance of the Configuration service.

<table>
<thead>
<tr>
<th>FROM ORCHESTRATOR VERSION</th>
<th>SERVICE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.5.1</td>
<td>IEndpointConfigurationService</td>
<td>Creates, reads, updates, and stores configurations as vRealize Orchestrator resources.</td>
</tr>
</tbody>
</table>
### 5.5.1 IEndpointConfiguration

Represents a single configuration that is stored as a vRealize Orchestrator resource.

### 5.5.1 IServiceRegistryAdaptor

An entry point to the IServiceRegistry. **NOTE** You should use this interface when the plug-in is not based on Spring.

### 5.5.1 IServiceRegistry

A service locator that provides Orchestrator services.

---

**Figure 1 shows a class diagram of a typical Orchestrator configuration that the Configuration service implements.**

---

Creating ConnectionInfo

`ConnectionInfo` is a simple plain old Java object (POJO) that holds all data required to create or configure a connection to a third-party system.

**NOTE** `ConnectionInfo` is not the actual connection, or scripting object, but rather the data that represents the object.

```java
public class ConnectionInfo {
    /*
     * Name of the connection
     */
    "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 + "]";
}

Creating the Configuration Change Listener

The configuration change listener is an extension point of the configuration persister, which is the interface that stores the data. With the configuration change listener, other components can subscribe to configuration events, such as update or delete a ConnectionInfo object.

Normally, the Orchestrator plug-ins store all live connections and sessions in the memory, as a self-managed cache. In this way, you can make sure that only a single instance of a connection or session exists. If another user updates or deletes this connection instance, by using the ConnectionPersister interface, the cache that stores the live connections or sessions, enters in an inconsistent state.

You can implement the observer pattern to prevent such a situation.

public interface ConfigurationChangeListener {
    /*
     * Invoked when the ConnectionInfo input is updated
     */
    void connectionUpdated(ConnectionInfo info);
    /*
     * Invoked when the ConnectionInfo input is deleted
     */
}
Connection Persister

ConnectionPersister is a class that creates, reads, updates, and deletes the ConnectionInfo objects. ConnectionPersister is a wrapper of IEndpointConfigurationService and hides the conversions between IEndpointConfigurationService and ConnectionInfo.

Defining the Configuration Persister Interface

```java
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 Default Connection Persister Interface

```java
@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 = "subscriptionId";
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() {
    //Initialize 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);
            result.add(connectionInfo);
        }
    } catch (Exception e) {
        //Log the exception
        log.error("Error retrieving configurations", e);
    }
    return result;
}
if (connectionInfo != null) {
    log.debug("Adding connection info to result map: " +
    connectionInfo);
    result.add(connectionInfo);
}
return result;
}

return result;
} catch (IOException e) {
    log.debug("Error reading connections.", e);
    throw new RuntimeException(e);
}
*/

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);
    }
}

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().toString());
            // 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);
    }
}
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

ConnectionRepository acts as local cache of a plug-in because it keeps all live connections. Any read operation that is related to a live connection must pass through the ConnectionRepository interface.

/*
 * 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 specific - 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);

        //Initialzing 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);
}

/*@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);
    }
}

/*@Override
public void connectionRemoved(ConnectionInfo info) {
    Connection live = connections.remove(info.getId());
    if (live != null) {
        live.destroy();
    }
}
}

The difference between ConnectionPersister and ConnectionRepository is that the persister manages only the ConnectionInfo POJOs that remain behind the live connections, whereas ConnectionRepository provides the same instance of a third-party connection. In other words, ConnectionRepository handles connections and ConnectionPersister handles ConnectionInfo objects.

ConnectionRepository provides two methods - findLiveConnection(...) and findAll(...). If you want to edit or delete a connection from the repository, you use the ConnectionPersister interface.

When the persister updates or deletes a connection, the change propagates to the repository because ConnectionRepository is subscribed to update or delete events that ConnectionPersister handles.

Connection

The Connection object that represents a live connection. There must be only a single Connection object instance per connection. The connection instances are stored in ConnectionRepository.

@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
     * initialized 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-initializes 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;
    }
}
Implement a Cluster Awareness of Configurations

vRealize Orchestrator instances can work in a cluster, which involves transferring configuration data between the clustered nodes. When you make a configuration change, for example, add an inventory object on one of the nodes in the cluster, this change does not automatically propagate to the workflow engines of the rest of the nodes.

The Orchestrator plug-in must contain a code that fulfills a set of requirements.

1. Detect that a change has been applied.

   The IEndpointConfigurationService.getVersion() method returns the current version of the configuration data in the database.

   The following code detects the change, by comparing the existing version to the one that is stored in the database.

   ```java
   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 is invoked every time an object that depends on the configuration data is accessed.

2. If the check detects a change, load configuration data.

   NOTE The Orchestrator platform caches the IEndpointConfigurationService.getVersion() method. The caching interval is different depending on the version of Orchestrator.

   a. If the Orchestrator version is 7.0.0 or later, the maximum caching interval is two heart-beat intervals.

   For example, if the heart-beat interval is five seconds and a configuration change occurs on Node1, calling the getVersion() method on the other nodes returns a cached value for a maximum period of 10 seconds.

   b. If the Orchestrator version is earlier than 7.0.0, the platform does not cache the version.

   In this case, caching is additionally provided and the TTL (Time To Live) interval is 10 seconds.

 IEndpointConfigurationService Implementation

During the plug-in development, you must implement IEndpointConfigurationService in such a way, that the IEndpointConfigurationService.getVersion() method can work on all versions of Orchestrator.

1. Make sure the plug-in depends on platform version 7.0.0 or later.

2. In the Maven pom.xml file, add a compile-time dependency to the o11n-plugin-tools artifact, so that the plug-in package includes the artifact.

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

3. Implement `IEndpointConfigurationService` depending on the type of the plug-in.

For Spring-based plug-ins, the name of the component is `cachingEndpointConfigurationService`.

a. The name the property must be `cachingEndpointConfigurationService`.

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

b. Add the application content XML to the plug-in.

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

For plug-ins that are not Spring-based, search the correct version, for example, in the `setServiceRegistry` method.

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

4. Reload the configuration data.

By fetching all data through `endpointConfigurationService`, find the changed entries on the other nodes.

5. Update the state of the inventory objects according to the loaded configuration data.

### Reading Configuration Files of Other Plug-Ins

<table>
<thead>
<tr>
<th>FROM ORCHESTRATOR VERSION</th>
<th>SERVICE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.5.1</td>
<td>IEndpointConfigurationService</td>
<td>Creates, reads, updates and stores configurations as vRealize Orchestrator resources.</td>
</tr>
<tr>
<td>5.5.1</td>
<td>IEndpointConfiguration</td>
<td>Represents a single configuration that is stored as a vRealize Orchestrator resource.</td>
</tr>
</tbody>
</table>

**NOTE** Some plug-ins rely on the configuration files of other plug-ins.

With `IEndpointConfigurationService`, you can retrieve a read-only configuration of another plug-in.

```java
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.setUrl(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) {

```java
    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

**Available in version 5.5.1 and later.**

The SSL service is a platform service that provides read-only access to the trust store files of the Orchestrator servers. Several workflows can write to the Orchestrator trust store. These workflows are part of the Orchestrator platform.

You can use these workflows in the configuration workflows of plug-ins that require establishing an SSL/TLS connection to the third-party system.

<table>
<thead>
<tr>
<th>WORKFLOW</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Library &gt; Configuration &gt; SSL Trust Manager &gt; Import a certificate from URL</td>
<td>Imports a trusted certificate from a remote URL. The certificate is imported with an automatically generated alias.</td>
</tr>
<tr>
<td>Library &gt; Configuration &gt; SSL Trust Manager &gt; Import a certificate from URL using proxy server</td>
<td>Imports a trusted certificate from a remote URL through a proxy server. The certificate is imported with an automatically generated alias.</td>
</tr>
</tbody>
</table>
Library > Configuration > SSL Trust Manager > Import a certificate from URL with certificate alias

Imports a trusted certificate from a remote URL. The certificate is imported with a specified alias.

Library > Configuration > SSL Trust Manager > Import a trusted certificate from a file

Imports a trusted certificate from a file. The file must be in a DER format.

Retrieve sslService from Service Registry

1. With a plug-in, you invoke the `getService()` method from `IServiceRegistry` and retrieve an `ISslService` interface.

   ```java
   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);
       }
   }
   ```

   NOTE If the plug-in is Spring-based, you can define the service dependencies, or auto-wire, `ISslService`.

2. After you retrieve the `sslService`, assign the service reference as a field.

   In this way, the plug-in does not search in the registry, every time it uses the service. `sslService` provides two interfaces - `SslContext` and `HostNameVerifier`.

   ```java
   public interface ISslService {
       SSLContext newSslContext(String protocol) throws NoSuchAlgorithmException;
       HostnameVerifier newHostNameVerifier();
   }
   ```

3. Create a new `SslContext` object and specify the desired protocol.

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

4. Create an `SSLSocketFactory` class.

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

   The factory class creates a socket with some specified options: `host`, `port`, `autoClose`, `localAddress`, `localPort`, and `localhost`.

5. Set `SSLSocketFactory` to an HTTPS connection.

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

6. When a URLConnection is open, use the HostNameVerifier interface.

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

**CAFÉ Gateway Service**

Available in version 5.5.2 and later.

The CAFÉ gateway service is a limited version of the SSO REST client service that exposes the RestClient class of the CAFÉ endpoints.

This service creates a preconfigured RestClient that communicates with the CAFÉ services in vRealize Automation. You can access the CAFÉ gateway service, no matter if the plug-in is Spring-enabled or not.

1. Access the service.
   a. If the plug-in is Spring-enabled:
      ```java
      //Object must be declared in the spring context
      public class MyDomainObject {
          @Autowired
          private CafeGateway cafeGateway;
      }
      
      b. If the plug-in is not Spring-enabled:
      ```
      ```java
      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;
          }
      }
      ```

2. Build a ScriptingObjectDefinition to read log events from vRealize Automation.

```java
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

Available in version 6.0.1 and later.

The SSO REST client service is a more extended version of the CAFÉ gateway service that exposes the RestClient class of the CAFÉ endpoints. With this service, you can specify the host and the root path of the REST client, instead of just communicating with a CAFÉ server that is configured as an authentication provider.

You can access the SSO REST client service, no matter if the plug-in is Spring-enabled or not.

1. Access the service.
   a. If the plug-in is Spring-enabled:

      // Object must be declared in the spring context
class MyDomainObject {
    @Autowired
    private SsoRestClientFactory restClientFactory;
  }

   b. If the plug-in is not Spring-enabled:

      class PluginAdaptor implements IServiceRegistryAdaptor {
        private IServiceRegistry registry;
        private SsoRestClientFactory ssoRestClientFactory;
        @Override
        public void setServiceRegistry(IServiceRegistry registry) {
          // Set the service registry
        }
      }
2. Create a SAML authentication session.

```java
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**

**Available in version 7.0.0 and later.**

The Solution authentication service provides access to the solution user token. The service does not provide a direct access to the token but creates an authentication object that you can use to construct a REST client that can send requests to the vRealize Automation APIs.

You can access the Solution authentication service, no matter if the plug-in is Spring-enabled or not.

3. Access the service.
   a. If the plug-in is Spring-enabled:

   ```java
   //Object must be declared in the spring context
   public class MyDomainObject {
   ```
@Autowired
private SolutionAuthenticationService solutionAuthenticationService;
}

b. If the plug-in is not Spring-enabled:

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_AUT

    return solutionAuthenticationService;
    }
}

Scripting Object Contributor Service

Available in version 7.0.0 and later.

With the Scripting object contributor service, plug-ins can register scripting objects at runtime, without having to restart the Orchestrator server.

<table>
<thead>
<tr>
<th>FROM ORCHESTRATOR VERSION</th>
<th>SERVICE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.0.0</td>
<td>ScriptingObjectsContributor</td>
<td>A service endpoint that registers scripting objects at runtime.</td>
</tr>
<tr>
<td>7.0.0</td>
<td>ScriptingObjectDefinition</td>
<td>A definition of a scripting object.</td>
</tr>
<tr>
<td>7.0.0</td>
<td>ScriptingAttributeDefinition</td>
<td>A definition of an attribute of a scripting object.</td>
</tr>
<tr>
<td>7.0.0</td>
<td>ScriptingMethodDefinition</td>
<td>A definition of a method of a scripting object.</td>
</tr>
<tr>
<td>7.0.0</td>
<td>ScriptingConstructorDefinition</td>
<td>A definition of a constructor of a scripting object.</td>
</tr>
</tbody>
</table>

1. Access the service.
   a. If the plug-in is Spring-enabled:

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

b. If the plug-in is not Spring-enabled:

```java
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;
    }
}
```

2. Contribute a scripting object with a single attribute.

```java
//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
cipherObjectContributor.contribute(Collections.singletonList(object));

Cipher Service

The Cipher service uses the Orchestrator encryption algorithms to encrypt and decrypt strings. You can use this service when a plug-in stores sensitive data.

1. Access the service.
   a. If the plug-in is Spring-enabled:

   ```java
   //Object must be declared in the spring context
   public class MyDomainObject {
       @Autowired
       private ICipher cipherService;
   }
   ```

   b. If the plug-in is not Spring-enabled:

   ```java
   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;
       }
   }
   ```
2. Invoke the encrypt and decrypt methods.

```java
public class MyCustomModelClass {
    private ICipher cipherService;

    // Encrypts a string
    cipherService.encrypt("text-to-encrypt");

    // Decrypts a vRO encrypted string
    cipherService.decrypt("..."); // the string for decryption
}
```

**NOTE** If the decryption fails, it will fall back to the legacy decryption algorithm that is available in earlier versions of Orchestrator.