Morpheus Plugin Documentation

Version 1.2.7

Table of Contents

Task Plugin
UI Extensions

Introduction

The Morpheus Plugin API is a Java 8 based library for creating Plugins that add functionality to Morpheus. The Plugin api supports implementing providers to Morpheus of the following types:

- UI Extensions
- Task Types
- IPAM
- DNS
- Approvals
- Cypher Modules
- Custom Reports
- Cloud Providers
- Network Providers

Release Notes

Note: Morpheus Plugin API is now 1.0! This means calls that are used will be supported for a longer period of time and given appropriate deprecation warnings when necessary. Morpheus Plugin API 1.2 requires a minimum Morpheus Version of `8.0.x`. Morpheus `7.0.x` still runs on Core version `1.1.x`

- **1.2.7**
  - **New Process Service API**
    - Introduced a new Process Service API for improved process management
    - Added support for internal process ID tracking during marshalling/unmarshalling operations
    - Enhanced process tracking capabilities between processing steps
  - **Backup Integration Enhancements**
    - Added support for backup plugin result exporting
    - Introduced custom backup integration detail views
    - Added ability to create custom backup provider tabs
    - Enhanced backup provider interface with view rendering capabilities
  - **Network Infrastructure Improvements**
    - Added complete Floating IP and Floating IP Pool services
    - Introduced NetworkResourceGroup and NetworkResourceGroupMember services
    - Added NetworkSwitch support with comprehensive service layer and models
    - Implemented ComputeServerInterfaceType services
  - **Cluster Package Support**

        - Added support for compute type packages
        - Implemented server group package functionality
        - Enhanced cluster management capabilities with package features
    - **API Improvements**
        - Added validateUsageOnDelete for NetworkTypes
        - Enhanced NetworkProvider with event subscription capabilities
        - Added support for filterServicePlans to CloudProvider
        - Enhanced StorageProviderVolumes interface with additional methods
        - Added new typed removeWorkload method and deprecated original implementation
        - Introduced suspendWorkload functionality to WorkloadProvisionProvider
        - Added ability to disable cookie usage in HTTP API Client
    - **Bug Fixes**
        - Fixed process ID tracking during marshalling/unmarshalling operations
        - Corrected fields handling in DatastoreEvent model
        - Resolved option types implementation issues in various providers
        - Fixed network provider synchronization issues
    - **Breaking Changes**
        - Deprecated existing interfaces in GenericIntegrationProvider
        - Introduced new event subscription models and interfaces (with backward compatibility)
        - Deprecated original removeWorkload implementation in favor of new typed method
- **1.2.5**
    - Added support for NetworkProvider plugins creating an integration without a cloud integration
    - Storage providers were moved to the storage context: morpheus.services.storageVolume ⇒ morpheus.services.storage.volume
    - Added ability for Network Providers to add UI tabs to the network integration detail page.
    - Added connection pooling to HTTP API Client
- **1.2.4**
    - Improvements to NetworkProvider types
    - Improvements to DatastoreTypeProviders
    - Beginnings of AffinityGroup support
- **1.2.3**
    - Enhancements to DatastoreTypeProviders for HPE VME/MVM.
    - FileCopyService capability advancements.
    - Various Dependency Updates

- **1.2.0**
  - OS Type Image support added to the various models to enable seeding and packaging of Os type based image associations.
- **1.1.9, 1.2.0**
  - Added DatastoreTypeProvider plugin support for MVM. This allows custom datastore plugins to interact with hosts and LUNs throughout the provisioning vm process.
- **1.1.6**
  - Finalized GenericIntegrationProvider and created generator
  - Finalized GuidanceRecommendationProvider and created generator
  - Finalized AnalyticsProvider and created generator
  - added endpoint for triggering agent upgrade
  - added endpoints for getting latest agent versions for current release
  - added stop/start/restart server actions
  - Handlebars and Asset reloading when a plugin is re-uploaded has been fixed.
  - Added additional endpoints for creating alarms
- **1.1.5**
  - Deprecated `callXmlApi()` method in `HttpApiClient`
  - Improve JSON handling in HttpApiClient
  - Added `content` and `contentFormatted` to `CatalogItemType`
  - Added `isSnapshot` to `BacupTypeProvider`
  - Added `BackupServer` to `ResourcePermission.ResourceType`
  - Added `statusPercent` to `VirtualImage`
- **1.1.4**
  - Added `removeInstance` to `LoadBalancerProvider.`
- **1.1.3**
  - Added missing image types to `ImageType` enum.
  - Moved `importWorkloadFacet` to `WorkloadProvisionProvider.`
  - Added methods for importing workloads and retrieving storage providers for storage buckets.
  - Added location getter on synchronous virtual image service.
  - Added `vipPool` to `NetworkLoadBalancerInstance.`
  - Includes all changes from 0.15.14
- **0.15.14**
  - Added editable, nameEditable, and deletable flags to `StorageVolumeType.`
  - Introduced secure boot, TPM, and credential guard to `VirtualImage.`

- Added IP normalization to `NetworkUtility`.

- **1.1.2**
  - Added support for synchronous services for backup storage.
  - Introduced methods to get bucket and bucket Karman provider for a backup.
  - Added TPM flag to `VirtualImage`.
  - Added new flags for default sync active state.
  - Updated forecast models.
  - Added interface method on provision providers for setting explicit descriptions on default instance types.
  - Includes all changes from 0.15.13
- **0.15.13**
  - Added `active` to `NetworkDomain`.
  - Introduced sync for `getCloudFileStreamUrl`.
  - Added `provisionRequiresResourcePool` flag to cloud types.
  - Added support for filtering networks and datastores during provisioning based on the selected resource pool.
- **1.1.1**
  - Added caching capabilities to `Qcow2InputStream`.
  - Introduced QCOW conversion tools/helpers for plugins.
  - Added various model maps and option types.
  - Enhanced inventory types and form field options.
  - Fixed network count and improved data query services.
  - Includes all changes from 0.15.11 and 0.15.12
- **0.15.12**
  - Deprecated unused host types.
  - Handled `maxCores` being null on plan lookup.
  - Allowed cloud plugins to set `canCreateNetworks` on cloud types.
- **0.15.11**
  - Made `getGlobalNetworkProxy` asynchronous and synchronous.
  - Added `isPlugin` property to `AccountResourceType`.
  - Added `getGlobalNetworkProxy` to the settings service.
- **1.0.6**
  - Added `addToDate` to `DateUtility`.
  - Introduced additional `callXmlApi` method options to match `callJsonApi` methods.
  - Includes all changes from 0.15.10
- **0.15.10**

- Added labels to compute server.
- Introduced appliance instance, execute schedule, and setting APIs.
- Added `systemImage` to `ImageLocation`.
- Added interface for scale providers.
- Supported IaC provisioning and resource mapping.
- Added support for workload metadata tag updates.

- **1.0.5**
  - Added Packages to compute type layout
  - Plugin API Services
  - Includes all changes from 0.15.9
- **0.15.9**
  - Added Packages to compute type layout
  - Updated snapshot management.
- **1.0.4**
  - Improvements to HTTPApiClient to support Certificate Auth as well as new methods for capturing response as a stream
  - Cloud Pool management support for cloud plugins and network associations
  - Added IPv6 CIDR to NetworkPool
  - Includes all changes from 0.15.8
- **0.15.8**
  - Improvements to HTTPApiClient to support Certificate Auth as well as new methods for capturing response as a stream
  - Cloud Pool management support for cloud plugins and network associations
  - Added IPv6 CIDR to NetworkPool
- **1.0.3**
  - Improvements to Model serialization
  - Additional method calls to support Amazon ScaleGroups
  - Added Backup Provider templates to generator
  - Includes all changes from 0.15.7
- **0.15.7**
  - Improvements to Model serialization
  - Additional method calls to support Amazon ScaleGroups
- **0.14.7**
  - Added missing method in NetworkUtility

- **1.0.2**
  - 1.0 Release with proper deprecation support!
  - Moved all rxjava calls to rxjava3 from rxjava2 (NOTE: This requires all plugins to be updated for 6.3.0 of morpheus)
  - Includes all changes from 0.15.6
- **0.15.6**
  - All Context Services now implement `MorpheusDataService`
  - Created SynchronousDataService equivalents for all asynchronous ones
  - Started HostProvider work for custom cluster types
  - New Task Provider format for simplification of making task plugins
  - Additional `Facets` for injecting functionality into various `ProvisionProviders`
  - ProvisionProvider classes split up based on type of provisioner. `WorkloadProvisionProvider`, `AppProvisionProvider`, `HostProvisionProvider`, and `CloudNativeProvisionProvider`.
- **0.15.5**
- **0.15.4**
  - Not released due to last minute issues
- **0.15.3**
  - Converting More Context Services to `MorpheusDataService` versions and deprecating old methods.
  - Deprecated direct service accessors on `MorpheusContext` in favor of `morpheusContext.getAsync()` for all the existing reactive services and `morpheusContext.getServices()` for all the synchronous counterparts.
  - Rename ComputeZonePool to CloudPool
  - Rename ComputeZoneRegion to CloudRegion
  - Rename ComputeZoneFolder to CloudFolder
  - Adding javadoc details to existing and new classes
  - Introducing `Facet` interfaces for adding additional functionality to `ProvisionProvider` implementations.
  - Starting to rename `IdentityProjection` objects to `Identity` for shorter naming convention.
  - New Base interfaces for ProvisionProvider based on if provisioning Compute or Cloud native resources or Apps.
  - **NOTE**: There are breaking changes in this plugin release for cloud plugins and likely more to come as we polish for 1.0 GA
- **0.15.2**
  - MorpheusDataService enhancements with added query methods.
  - Deprecated Service access directly on `MorpheusContext` in favor of accessing thru sub classes i.e. `morpheusContext.getAsync().getService()`.
  - Began adding non-reactive synchronous service access via `morpheusContext.getServices().getService()`
  - Improved javadoc for `DataQuery` and `DataService` methods.
- **0.15.1**

    - Moved most Providers new packages folder `com.morpheusdata.core.providers`
    - Deprecated `OptionSourceProvider` in favor of new `DatasetProvider`
        - Enables scribe export/import object reference mapping and hcl data lookup as well
    - Service Consistency work in the `MorpheusContext.`
        - Created new `MorpheusDataService` interface reference that allows for using dynamic db queries and object marshalling into the core/api models.
    - New `StorageProvider` work began for abstracting various storage providers within morpheus.
    - Enhanced `NetworkProvider` to support `Router` and `SecurityGroup` representations.
- **0.15.0**
    - Filling in more Models and Cloud representations.
- **0.14.4**
    - Fixed an issue where the BackupProvider wasn't marshalled to the cloud on option sources.
- **0.14.3**
    - Filling in more Models and Cloud representations.
    - Completed Localization support. Plugins now can be fully localized in both server side, and client side rendering. Guide provided as well.
- **0.14.2**
    - Filling in more Models and Cloud representations.
    - Added OptionType support for the `hidden` HTML Input.
- **0.14.1.**
    - Filling in more Models and Cloud representations.
- **0.14.0**
    - Filling in Cloud related gaps as we work to provide full cloud provider plugin support
    - F5 Load Balancer support added and full abstractions for the `LoadBalancerProvider.`
- **0.13.4**
    - Backup Plugin Support Added
    - Cloud Plugin Coverage Improved
    - DNS Plugins can now function standalone
    - HTTP ApiClient now uses CharSequence for GString compatibility
    - Improved Javadoc
    - IPAMProvider Interface removed unnecessary methods
    - Task Type Icons now use a getIcon() method on the Provider
    - Network Pool Objects added IPv6 information (more to come)
    - Context Services for Syncing additional cloud object types (such as Security Groups)

    - Various other bug fixes and improvements on the road to 1.0.0
    - Bump JVM Compatibility minimum to 1.11 (jdk 11)
- **0.13.1**
    - Added Credential Providers support as well as significant CloudProvider refactoring (more to follow)
- **0.12.5**
    - Task Providers now have a hasResults flag for result variable chaining.
- **0.12.4**
    - IPAM NetworkPoolType filters for handling multiple pool types in one integration.
    - Deprecated reservePoolAddress from IPAMProvider as its no longer needed.
    - Added typeCode to the `NetworkPoolIdentityProjection`.
    - Added `{{nonce}}` helper to handlebars tab providers for injecting javascript safely within the Content Security Policies in place.
- **0.12.3**
    - Simplification and Polish if IPAM/DNS Interface Implementations (need Morpheus 5.4.4+).
    - Added new ReportProvider helper for easier management of db connection use `withDbConnection { connection → }`.
- **0.12.0**
    - Cloud Provider Plugin Critical Fixes (WIP).
    - Added Plugin settings.
- **0.11.0**
    - Cloud Provider Plugin Support.
    - UI Nonce token attribute added for injecting javascript securely and css.
    - Network Provider Plugin support. Create providers for dynamically creating networks and network related objects.
- **0.10.0**
    - Custom Report Type Providers have been added.
- **0.8.0**
    - Overhauled DNS/IPAM Integrations, Reorganized contexts and standardized formats. \
    - Added utility classes for easier sync logic.
    - Custom reports, Cloud Providers, Server Tabs, and more.
    - Only compatible with Morpheus version `5.3.1` forward.
- **0.7.0**
    - Please note due to jcenter() EOL Don't use 0.7.0
- **0.6.0**
    - Primary Plugin target base version for 5.2.x Morpheus Releases

Getting Started

Developing plugins for Morpheus is a bit different than simply creating custom tasks or custom library items in Morpheus. It requires some programming experience. We should preface with the fact that Morpheus runs on top of Java. It is primarily written in a dynamic language called Groovy (https://groovy-lang.org) and is based on Groovy version 3.0.7 currently running within Java 11 (openjdk). The provided plugin api dependency provides a common set of interfaces a developer can implement to dynamically load functionality into the product. These developers are free to develop plugins in native Java or leverage the groovy runtime. Most examples provided are developed and sampled in groovy. It is a great dynamic language that is much less verbose and easier to understand than native java based languages. Plans for supporting additional languages are in the works. Most notably kotlin is being looked into as an alternative development platform. jRuby is also a viable language that can be used as the runtime is included with Morpheus. The plugin architecture is designed based on reactive models using `rxJava2`. This library follows ReactiveX (http://reactivex.io/) models for Observer pattern based programming. One unique thing about Observer pattern programming is to remember when writing a call to an `Observable` the code is not executed until the final `subscribe` call. If this call is missing, the code will never execute.

NOTE: `rxJava3` is a different project build and cannot be used yet in place of `rxJava2`.

The plugin pipeline leverages Gradle (https://gradle.org) as the build tooling. Gradle is a cross-platform programmatic build tool that is very commonly used and is most notably also used in the android space. To begin make sure your development environment has Gradle 6.5 (at least but if using newer Gradle make sure you are familiar with the definition changes needed) as well as Java 11 (if using openjdk over 11 make sure target compatibility is set to 1.11 within your project).

NOTE: Currently Gradle 7.x is recommended as we upgrade support to Gradle 8

The morpheus-plugin-core (https://github.com/gomorpheus/morpheus-plugin-core) git repository is structured such that sample plugins exist underneath the `samples` directory. They are part of a multi-project gradle project as can be defined in the root directories `settings.gradle`. This adds a bit of complexity and should be ignored when developing ones own plugin. Simply start with a blank project folder and a simple `build.gradle` as demonstrated below.

The structure of a Plugin is a typical "fat jar" (aka shadowJar). The plugin will include `morpheus-plugin-api` along with all the dependencies required to run. Though not required, it is often conventional to use the groovy programming language when developing plugins as can be seen in the samples. Morpheus is based on the groovy runtime and therefore allows full use of groovy 3.0.x.

The Structure of a project often starts with a gradle build along with a `Plugin` class implementation and a manifest that points to this class. This is the entrypoint for the plugin where all metadata about the plugin is defined as well as all `Providers` offered by the plugin are registered.

Gradle project

Create a new project with a `build.gradle` file. We will use the shadowjar plugin to create our "fat jar"

build.gradle

GROOVY

```groovy
plugins {
    id "com.bertramlabs.asset-pipeline" version "4.3.0"
    id "com.github.johnrengelman.shadow" version "6.0.0"
}

buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath "com.morpheusdata:morpheus-plugin-gradle:0.14.3"
    }
}

apply plugin: 'com.morpheusdata.morpheus-plugin-gradle'
apply plugin: 'java'
apply plugin: 'groovy'
apply plugin: 'maven-publish'

group = 'com.example'
version = '1.0.0'

sourceCompatibility = '1.11'
targetCompatibility = '1.11'

ext.isReleaseVersion = !version.endsWith("SNAPSHOT")

repositories {
    mavenCentral()
}

configurations {
    provided
}

dependencies {
    provided 'com.morpheusdata:morpheus-plugin-api:0.12.5' //use 0.13.4 for 5.5.x
    provided 'org.codehaus.groovy:groovy-all:3.0.9'

    /*
     When using custom libraries, use the gradle `implementation` directive
     instead of `provided`.
     This will allow shadowJar to package the library into the resulting plugin and keep it
     isolated within the same classloader.
     */
}

tasks.assemble.dependsOn tasks.shadowJar
```

To let the Morpheus plugin manager know what class to load you must specify the class in jar's manifest:

build.gradle

GROOVY

```groovy
jar {
    manifest {
        attributes(
                'Plugin-Class': 'com.example.MyPlugin', //Reference to Plugin class
                'Plugin-Version': archiveVersion.get() // Get version defined in gradle
                'Morpheus-Name': 'Plugin Name',
                'Morpheus-Organization': 'My Organization',
                'Morpheus-Code': 'plugin-code',
                'Morpheus-Description': 'My Plugin Description',
                'Morpheus-Logo': 'assets/myplugin.svg',
                'Morpheus-Logo-Dark': 'assets/myplugin-dark.svg',
                'Morpheus-Labels': 'Plugin, Stuff',
                'Morpheus-Repo': 'https://github.com/myorg/myrepo',
                'Morpheus-Min-Appliance-Version': "5.5.2"
        )
    }
}
```

When writing plugin code, it is important to note a typical groovy/java project folder structure

ls -R

BASH

```
./
.gitignore
build.gradle
src/main/groovy/
src/main/resources/renderer/hbs/
src/main/resources/i18n
src/main/resources/scribe
src/main/resources/packages
src/test/groovy/
src/assets/images/
src/assets/javascript/
src/assets/stylesheets/
```

Be sure to configure your `.gitignore` file to ignore the `build/` directory which appears after performing your first build.

Most of the folder structure can be self-explanatory if familiar with groovy/java. Project packages live within `src/main/groovy` and contain source files ending in `.groovy`. View resources are stored in the `src/main/resources` subfolder and vary depending on the view renderer of choice. While static assets like icons or custom javascript live within the `src/assets` folder. This is handled by the Asset Pipeline (http://www.asset-pipeline.com) plugin. View rendering and static assets will be covered in more detail later.

Building a project is as simple as calling

BASH

```
./gradlew shadowJar
```

NOTE: If the gradle wrapper does not yet exist, simply run `gradle wrapper` within the root of the project to generate the wrapper.

The resulting jar will exist, by default, in the `build/libs` directory of the project.

Plugin Class

Following the example above, create your plugin class under `src/main/java/com/example/MyPlugin.java`

Your plugin must extend the `com.morpheus.core.Plugin` class:

MyPlugin.java

JAVA

```
import com.morpheus.core.Plugin;

class MyPlugin extends Plugin {

        @Override
        void initialize() {
                this.setName("My Custom Tabs Plugin");
                CustomTabProvider tabProvider = new CustomTabProvider(this, morpheus);
                this.registerProvider(tabProvider);
        }
}
```

Here we see a basic plugin that initializes some metadata (the Plugin Name) and registers a custom tab provider.

Registering Providers

A plugin may contain any number of Plugin Providers. A Plugin Provider contains the functionality of your plugin such as a Custom Tab, IPAM, Backup Provider, etc.

There are provided classes such as `TaskProvider`, `ProvisionProvider`, `ApprovalProvider`, `ReportProvider`, `InstanceTabProvider`, `ServerTabProvider`, and others to get you started building your provider. For example the `InstanceTabProvider` provides a renderer, show method, and security checking method to make it easy to build your own custom tab on the instance page.

Providers are registered in your plugin during the `initialize()` method call within the `Plugin` class as can be seen in the `MyPlugin.java` sample seen above.

Settings

As of 0.8.0 Plugins can now have settings that can be applied globally after installing a plugin. Some users may use this to configure an integration to a third party service like Datadog, or affect how providers may behave based on some setting. These can be set after uploading a plugin in `Administration` → `Integrations` → `Plugins`.

Settings, much like any other form aspect of a plugin, can take advantage of `OptionType` and `OptionProvider` entities to configure how the options are presented to the user and what options are available to choose from.

GROOVY

```
    class MyPlugin extends Plugin {
        /**
         * Returns a list of {@link OptionType} settings for this plugin.
         * @return this list of settings
         */
        public List<OptionType> getSettings() { return this.settings; }
    }
```

Implementing the above getter method allows one to specify the form option-types of settings that could be saved by the user.

Fetching setting values for use in a provider can easily be accomplished via the `morpheusContext`. There is a method that returns a JSON String of the setting values that is up to the plugin developer to deserialize called `morpheusContext.getSettings(Plugin plugin)`. Simply pass the plugin class instance to it and an rxJava `Single<String>` is returned.

GROOVY

```
String pluginSettings = morpheus.getSettings(this.plugin).blockingGet()
def pluginDeserialized = new JsonSlurper().parseText(pluginSettings)
```

Contexts / Services

The Morpheus context allows you to interact with, query and save data with Morpheus. It is organized into several sub service classes to perform operations that may involve database calls or calling common methods within the Morpheus core that are not available directly. Some calls may be as simple as `listById` or `save`. But they can also be as complex as `executeSshCommand` or `executeWinrmCommand`. All calls within a Morpheus Context implement RxJava2 conventions. For details on how to program in Reactive syntax and RxJava concepts please see the documentation site http://reactivex.io/.

When interacting with various subcontexts it is helpful to know there is a common guideline on method names involving database calls. These common method names include:

- `listIdentityProjections`
- `listById`
- `listBy*`
- `get`
- `findBy*`
- `remove`
- `create`
- `save`

There are, of course, exceptions and non ORM related methods that also may exist in certain services that provide common helper methods. For a full listing of the various service classes please check the API Docs and look at the MorpheusContext (https://developer.morpheusdata.com/api/com/morpheusdata/core/MorpheusContext.html) class. There are several in line with general ORM calls as well as some related to certain actions.

Models

When interacting with Morpheus you must use the models provided in `com.morpheusdata.model.*`. This allows for the context to be strongly typed. These Models have a few conventions that you might like to know about. Firstly, models that are often synced from a cloud provider or integration of any kind often inherit from their base class known as an `IdentityProjection` which also inherits from `MorpheusModel`.

For Example: the `NetworkDomain` model inherits `NetworkDomainIdentityProjection` which in turn inherits `MorpheusModel`.

The `IdentityProjection` contains a subset of properties that are typically used to match the object with its equivalent on the other side of an Api implementation. This is typically persisted in the `externalId` property of the object. Some objects also leverage `uniqueId` as well.

It is also recommended to use the getter and setter methods on the Models (which should be strictly required in the near future) so as to ensure the dirtyProperties map is updated appropriately. This will be used in future distributed worker installations to reduce bandwidth in transmission of data updates back to Morpheus.

Data Services

As the Morpheus Plugin API has progressed a new concept was created in 0.15.x. It was determined a consistent means for querying Morpheus data objects needed to be made across the board in all of the `MorpheusContext` accessible services. Previously, there were only a fixed set of options per Model that one might have wanted to query. But with the development of the `DataService` model and the new `DataQuery` object, a lot more power is available to the developer.

These services are now commonly used in various sync related activities with `SyncTask` as well as can be used with custom reporting or the new `DataSetProvider` concepts.

MorpheusDataService.java

```java
public interface MorpheusDataService<M extends MorpheusModel, I extends MorpheusModel> {
        Observable<M> listById(List<Long> ids);

        Observable<M> list(DataQuery query);

        Observable<Map> listOptions(DataQuery query);

        Maybe<M> find(DataQuery query);


        Single<DataQueryResult> search(DataQuery query);
}
```

Above you will see a common set of methods for querying data. The important part of these methods is to note that they all take a `DataQuery` object. This object allows you to scope a query to a `User` or an `Account` within morpheus as well as pass in complex query operators.

For Example:

```groovy
def usageClouds = morpheusContext.async.cloud.list(
        new DataQuery().withFilters(
                new DataOrFilter(
                        new DataFilter("externalId","in",usageAccountIds),
                        new DataFilter("linkedAccountId","in",usageAccountIds)
                )
        )
).toList().blockingGet()
```

The above query looks for all clouds that contain a set of usage account ids via either the `externalId` property or the `linkedAccountId` property.

As can be seen above, queries can be nested with `DataOrFilter` or `DataAndFilter` combinations to build complex queries. More details of this also exist on the `DataQuery` java doc.

NOTE: The DataService also provides `save`, `create`, `remove` methods of a consistent nature. Please refer to the APIDoc / javadoc for descriptions on how to use these methods.

In the past, when making custom reports, it was common to use the direct database connection and query the morpheus data set directly. This carried with it some risks as these tables are not strictly documented and some data is in an encrypted state. By utilizing data services (where possible), however, the object models are marshalled into a properly

documented format and fields that may have been previously inaccessible, due to encryption, are now available for use.

Synchronous Data Services

In an effort to make plugin development easier, it was decided that a counterpart service would be made for non rxjava / reactive calls. We call this the `SynchronousDataService`. This service provides the exact same methods as the `MorpheusDataService`, however, it does not require an Observable subscription or `blockingGet()`.

These services can be accessed via the contexts `morpheusContext.getServices()` directive as opposed to `morpheusContext.getAsync()`. They are useful for things that you know are going to block anyway, such as perhaps a UI Page render.

It is still recommended, when performing sync operations of a cloud, or generating a custom report to utilize the asynchronous counterparts for best performance.

NOTE: As of 0.15.3: Not All Async services have been converted yet and several may still be missing.

Syncing Data

Many of the Morpheus Provider types have a method to periodically refresh and sync data into Morpheus. This can be useful for representing core concepts or doing brownfield discovery for things like `ComputeServer` or `NetworkPool` objects. Also, it may be useful for syncing in user presented options in a dropdown or typeahead during a particular operation (Generic Data for this we can store n `ReferenceData`).

Over the course of several years, Morpheus has developed an optimal way to efficiently sync data from a remote endpoint into the appliance. This has to take into account network bandwidth, memory , cpu, and load on the target api. Morpheus also created a helper class for making sync operations simple and consistent called the `SyncTask` (javadoc available).

Below is an example that syncs available amazon regions in the AWS Cloud to the CloudRegion table.

GROOVY

```
Observable<CloudRegionIdentity> domainRecords =
morpheusContext.async.cloud.region.listIdentityProjections(cloud.id)
SyncTask<CloudRegionIdentity, Region, CloudRegion> syncTask = new SyncTask<>(domainRecords,
regionResults.regionList as Collection<Region>)
syncTask.addMatchFunction { CloudRegionIdentity domainObject, Region data ->
        domainObject.externalId == data.getRegionName()
}.onDelete { removeItems ->
        removeMissingRegions(removeItems)
}.onUpdate { List<SyncTask.UpdateItem<CloudRegion, Region>> updateItems ->
        updateRegions(updateItems)
}.onAdd { itemsToAdd ->
        addMissingRegions(itemsToAdd, this.@cloud.account)
}.withLoadObjectDetailsFromFinder { List<SyncTask.UpdateItemDto<CloudRegionIdentity, Region>> updateItems ->
        morpheusContext.async.cloud.region.listById(updateItems.collect { it.existingItem.id } as List<Long>)
}.start()
```

NOTE: The above example does not show that the api query to get all regions was performed above.

A `SyncTask` is capable of taking the `Identity` objects of a class (small form of a `Model` that contains just the important identification fields used for matching) and comparing them against the api results list. This is done via the `addMatchFunction` as seen above.

NOTE: It is possible to have a chain of match functions as secondary fallbacks by simply adding another to the chain.

Objects that do not have a match from the api are sent in batches to the `onAdd` for creation of Morpheus `Model` objects and consequently the inverse is true for models that are not found within the api results as these are sent to `onDelete`.

Objects that do match are a bit different. Firstly, in order, to check the object for changes it must first be fully loaded from its `Identity`. This can be seen in the `withLoadObjectDetailsFromFinder` call. This translates those Identity classes into their fully loaded objects in batches of 50 at a time (not seen as this is magically done by the SyncTask). Once these objects are fully loaded they are passed down into the `onUpdate` where they can be compared.

Although not seen above, it is best to bulk create or bulk save morpheus changes in these chunked add and update methods. This is done using the `MorpheusDataService.bulkCreate` or `MorpheusDataService.bulkSave` implementations on the context.

Sync methods such as this exist in several provider types beyond just clouds. IPAM Providers, DNS Providers, backups and others all contain endpoints where these sync operations can be performed.

HTTP Routing

A plugin may register an endpoint or endpoints to respond with `html` or `json`. To register your routes in your plugin you must implement the `PluginController` class.

Models

Incoming requests come with a `ViewModel` object populated with the http request & response details.

Example

GROOVY

```
class MyPluginController implements PluginController {
        List<Route> getRoutes() {
                [
                        Route.build("/myPrefix/example", "html", Permission.build("admin", "full")),
                        Route.build("/reverseTask/json", "json", Permission.build("admin", "full"))
                ]
        }

        def html(ViewModel<String> model) {
                return HTMLResponse.success("Some Text")
        }

        def json(ViewModel<Map> model) {
                Map simpleMap = [serverid: "abc-123", other: model.object.someData]
                return JsonResponse.of(simpleMap)
        }
}
```

Route provides a builder to allow your plugin to easily build a route with permissions. It takes the `url`, the `method` in this class to call, and a list of `permissions` which can be built with the `Permission` builder.

The route can either return:

- `HTMLResponse` - simple text, or a full rendered view.
- `JsonResponse` - an object rendered as json.

After creating a `PluginController`, register it in your plugin like so:

MyPlugin.java

JAVA

```
        @Override
        void initialize() {
                this.setName("My Custom Task Plugin");
                CustomTaskProvider taskProvider = new CustomTaskProvider(this, morpheusContext);
                this.pluginProviders.put(taskProvider.providerCode, taskProvider);

                this.controllers.add(new MyPluginController());
        }
```

HTTP API Client

The `morpheus-plugin-api` library comes with a utility for facilitating quick and easy API calls to external integrations. This is called teh `HttpApiClient` and replaces the use of the `RestApiUtil`.

The developer is, of course, able to utilize any HTTP or SDK/API client they choose within the java ecosystem if so desired. However, this client provides some common functions such as SSL Validation, Throttling, Keep-Alive, Etc. and is based on the Apache HTTP Client.

TIP Reuse the same client instance when dealing with periodic refresh methods related to caching for best performance. Be mindful that throttling might be necessary to slow down the calls to the service.

Example

Below is an example API Call taken from the `Infoblox` plugin.

GROOVY

```
import com.morpheusdata.core.util.HttpApiClient

HttpApiClient infobloxClient = new HttpApiClient()
try {
    def results = infobloxClient.callJsonApi(serviceUrl, apiPath, poolServer.serviceUsername,
poolServer.servicePassword, new HttpApiClient.RequestOptions(headers:['Content-Type':'application/json'],
queryParams:pageQuery, ignoreSSL: poolServer.ignoreSsl), 'GET')
} catch(e) {
    log.error("verifyPoolServer error: ${e}", e)
} finally {
    infobloxClient.shutdownClient()
}
return rtn
```

There are several methods to call the api and automatically handle certain payload formats such as `callJsonApi`, `callXmlApi`, or plain `callApi`.

Options can be passed relating to the request via the `HttpApiClient.RequestOptions` object. Please refer to the java api doc for available options.

TIP Remember to always shutdown the client after it is used to clean up the connection manager in a `try{} finally{}` type block

Views

Plugins may render html sections such as adding a tab to different areas of Morpheus which you can populate with your own content. By default a Handlebars template provider is provided by the Plugin Manager. If you wish to use your own template engine you may implement `com.morpheusdata.views.Renderer` interface.

Rendering a view

Views are stored in your plugin at `src/main/resources/renderer/<plugin scope>/<your view>.hbs`. Many plugin providers provide a convention to store views that will be rendered.

If you wish to render and return html manually you can call the renderer directly:

```java
getRenderer().renderTemplate("prefix/someview", model);
```

Do not provide the suffix (.hbs) - you may also pass a `com.morpheusdata.views.ViewModel` into the view to use when rendering the html.

Asset helper

The template engine can also process static assets such as images, css, and javascript for you. Place your assets under `src/assets/{plugin-code}`. To include them in the plugin template you can use the `{{ asset }}` handlebars helper as so:

```html
<script src="{{asset "/instance-tab.js"}}"></script>
<img src="{{asset "/some/image.png"}}" />
```

Localization

Morpheus Plugins support using i18n localization properties. These are string maps for representing various sections of text in your plugin (or in morpheus itself) by a localized way. Creating a `messages.properties` file in the directory `src/main/resources/i18n` will allow you to set some key value maps.

For Example `messages.properties`:

```properties
com.morpheusdata.label.hello=Hello
```

or for Spanish (ES) `messages_es.properties`:

```properties
com.morpheusdata.label.hello=Hola
```

These can be leveraged in the handlebarsRenderer for server side rendering via the `{{i18n}}` helper like so:

```handlebars
<strong>{{i18n 'com.morpheusdata.label.hello'}}</strong>
```

Or they can also be used in javascript rendering (like in dashboard widgets or other areas):

```javascript
var helloString = $L({code: 'com.morpheusdata.label.hello', default: 'Hello'});
```

If a matching string exists for the current browser locale, it will be used, if not it will fall back to the default `messages.properties` file.

Localizing the Entire Application

Plugin localization files are global. This means, if a language pack did not exist for the entire Morpheus Appliance for a specific locale and a developer/partner wanted to make one as a plugin, they could. The plugins properties are automatically checked system-wide. It is even possible to override single labels in the morpheus appliance if so desired.

For information on localizing Morpheus, please refer to the Morpheus Crowdin Localization Plugin (https://share.morpheusdata.com/crowdin-localization-plugin/about)

DataSet Providers

A DataSetProvider is a server side call to load a dynamic data for custom form inputs, as well as data references with scribe templates. A dataset can feed data to dropdown lists, multiselect components, and typeahead components. Datasets can be very useful when designing task types that have custom options or even custom catalog item layouts. In the past, this type of data was provided with an `OptionSourceProvider` which contained multiple definitions in a single provider, but had its own limitations and did not provide sufficient documentation on the data provided. The `DataSetProvider` improved dynamic data definitions by: isolating one dataset per provider, including associative information so that it can be used for export/import scribe functionality, and including documentation on the dataset within the dataset definition.

Getting Started

Create an implementation of a `DataSetProvider` or with some convenience methods an `AbstractDataSetProvider`.

When defining an `OptionType` set the `optionSource` property to the `getKey()` defined in the dataset provider. Avoid naming conflicts with other plugins with unique method name or isolate the dataset by using a namespaces. A namespace is set on the `namespace` property of the `DatasetInfo` in `getDataset()`. Alternatively, the `getNamespace()` method can be implemented or overridden. When a dataset is namespaced the `optionSourceType` of an `OptionType` or `OptionSource` should correlate to the dataset's namespace.

> **TIP** Use a unique namespace for your DataSetProvider to isolate from other plugins.

GROOVY

```
OptionType ot4 = new OptionType(
name: 'Region',
code: 'google-plugin-region',
fieldName: 'googleRegionId',
optionSourceType: 'google', // Note: this references the dataset namespace.
optionSource: 'googlePluginRegions', // Note: this references the dataset key.
displayOrder: 3,
fieldLabel: 'Region',
required: true,
inputType: OptionType.InputType.SELECT,
dependsOn: 'google-plugin-project-id',
fieldContext: 'config'
)
```

DatasetProvider examples can be found in the example plugin repository (https://github.com/gomorpheus/morpheus-plugin-dataset-examples/tree/main/src/main/groovy/com/morpheusdata/datasets) on GitHub.

Option Sources

NOTE These have been replaced by `DataSet Providers` as of 0.15.x.

DEPRECATED: An Option source is a server side call to load a dynamic dataset for custom form inputs. These normally feed dropdown lists, multiselect components, and typeahead components. These can be very useful when designing task types that have custom options or even custom catalog item layouts.

Getting Started

To get started, simply create an implementation of an `OptionSourceProvider`. The primary implementation method is called `getMethodNames` and just returns a list of methods on the class that are accessed via API as datasets.

GROOVY

```
class GoogleOptionSourceProvider implements OptionSourceProvider {
@Override
    List<String> getMethodNames() {
        return new ArrayList<String>(['googlePluginProjects', 'googlePluginRegions', 'googlePluginZonePools',
'googlePluginMtu'])
    }

    def googlePluginProjects(args) {
        Map authConfig = getAuthConfig(args)
        def projectResults = []
        if(authConfig.clientEmail && authConfig.privateKey) {
            def listResults = GoogleApiService.listProjects(authConfig)
            if(listResults.success) {
                projectResults = listResults.projects?.collect { [name: it.name, value: it.projectId] }
                projectResults = projectResults.sort { a, b -> a.name?.toLowerCase() <=> b.name?.toLowerCase() }
            }
        }
        projectResults
    }
}
```

Above is an example option source registered for the Google Cloud Plugin. **NOTE:** Every method must have a singular input argument of type `Object` (which is the default in Groovy). In reality, it is a Map containing passed in params from the api call that can be referenced as well as the current user object. It is also worth noting that the return type expected is a `List<Map<String,String>>` whereby the properties on the map are of keys `name,value`.

Now, when defining your `OptionType` simply set your `optionSource` property to the field name defined in your provider. It is important not to conflict names with other plugins so please try to use a unique method name.

> **TIP** Use a Unique Method name that will not interfere with other plugins the user may load.

GROOVY

```groovy
OptionType ot4 = new OptionType(
                                name: 'Region',
                                code: 'google-plugin-region',
                                fieldName: 'googleRegionId',
                                optionSource: 'googlePluginRegions', //Note the Option Source Defined here.
                                displayOrder: 3,
                                fieldLabel: 'Region',
                                required: true,
                                inputType: OptionType.InputType.SELECT,
                                dependsOn: 'google-plugin-project-id',
                                fieldContext: 'config'
                        )
```

Credential Provider Inputs

During the 5.4.x and 5.5.x release of Morpheus. Credentials were introduced. It became possible to store integration credentials externally or decoupled from a specific integration. This is great for service accounts! As a result a new option type was created to allow them to be used for custom plugin integrations.

Getting Started

Most of the time, this setup will be used when defining available OptionTypes in the various Provider types that have them. For example, an `IPAMProvider` as a method for `getIntegrationOptionTypes()`. The idea is to also support local credential inputs like the simple `serviceUsername` and `servicePassword` fields. To do this simply add the flag `localCredential:true` to those fields so the system knows, when using local credentials, to show those fields. An example of a credential provider being used can be seen here.

GROOVY

```
class InfobloxProvider implements IPAMProvider, DNSProvider {
@Override
    List<OptionType> getIntegrationOptionTypes() {
        return [
                new OptionType(code: 'infoblox.serviceUrl', name: 'Service URL', inputType:
OptionType.InputType.TEXT, fieldName: 'serviceUrl', fieldLabel: 'API Url', fieldContext: 'domain', placeHolder:
'https://x.x.x.x/wapi/v2.2.1', helpBlock: 'Warning! Using HTTP URLS are insecure and not recommended.',
displayOrder: 0, required:true),
                new OptionType(code: 'infoblox.credentials', name: 'Credentials', inputType:
OptionType.InputType.CREDENTIAL, fieldName: 'type', fieldLabel: 'Credentials', fieldContext: 'credential',
required: true, displayOrder: 1, defaultValue: 'local',optionSource: 'credentials',config: '{"credentialTypes":
["username-password"]}'),

                new OptionType(code: 'infoblox.serviceUsername', name: 'Service Username', inputType:
OptionType.InputType.TEXT, fieldName: 'serviceUsername', fieldLabel: 'Username', fieldContext: 'domain',
displayOrder: 2,localCredential: true),
                new OptionType(code: 'infoblox.servicePassword', name: 'Service Password', inputType:
OptionType.InputType.PASSWORD, fieldName: 'servicePassword', fieldLabel: 'Password', fieldContext: 'domain',
displayOrder: 3,localCredential: true),

                ...
}
```

Note the `infoblox.credentials` option type and its use of the type `OptionType.InputType.CREDENTIAL`. The `optionSource` is critical as well as the list of possible credential types seen in the `config` block.

There are other credential types available that will be populated in this documentation later.

Using Credentials in your plugin.

When making remote calls to the integration code it is important to reference the credential data correctly. For example, it is no longer simply a matter of referencing `poolServer.serviceUsername` or `poolServer.servicePassword`. Instead it is important to check the `credentialData` map on the `AccountIntegration`, `Cloud`, or `NetworkPoolServer`.

GROOVY

```
    results = client.callApi(serviceUrl, apiPath, poolServer.credentialData?.username ?:
poolServer.serviceUsername, poolServer.credentialData?.password ?: poolServer.servicePassword, new
HttpApiClient.RequestOptions(headers:['Content-Type':'application/json'], ignoreSSL:
poolServer.ignoreSsl,body:body), 'POST')
```

An example can be seen above where the credentialData is first checked.

Testing

We recommend Spock (http://spockframework.org/) for easily testing your plugin. The interfaces can easily be mocked and stubbed to allow you to test your integrations without a running Morpheus instance.

Example

GROOVY

```
class InfobloxProviderSpec extends Specification {
    @Shared MorpheusContext context
    @Shared InfobloxPlugin plugin
    @Shared InfobloxAPI infobloxAPI
    @Shared MorpheusNetworkContext networkContext
    @Subject @Shared InfobloxProvider provider

    void setup() {
        // Create a Mocks of the Morepohus contexts you will use
        context = Mock(MorpheusContextImpl)
        networkContext = Mock(MorpheusNetworkContext)
        context.getNetwork() >> networkContext
        plugin = Mock(InfobloxPlugin)
        infobloxAPI = Mock(InfobloxAPI)

        // Create the actual provider to unit test
        provider = new InfobloxProvider(plugin, context, infobloxAPI)
    }

    void "listNetworks"() {
        given: "A pool server"
        def poolServer = new NetworkPoolServer(apiPort: 8080, serviceUrl: "http://localhost")
        // Here we are stubbing the actual API call to infoblox, but we could create a integration test by
actually providing the real infoblox API class instead of a mock.
        infobloxAPI.callApi(_, _, _, _, _, _) >> new ServiceResponse(success: true, errors: null ,
content:'{"foo": 1}')

        when: "We list the networks"
        def response = provider.listNetworks(poolServer, [doPaging: false])

        then: "We get a response"
        response.size() == 1
    }
}
```

As you can see, implementing unit and integration testing for your plugins can be done easily with Spock. Of course any other JVM unit testing framework should work as well.

Seeding data during plugin installation

If you need to ensure that certain data (e.g. layouts, plans, etc.) are added to the database when your plugin is installed, you can use this process to "seed" that data. Morpheus consumes and produces HCL formatted data (https://github.com/hashicorp/hcl/tree/main?tab=readme-ov-file#information-model-and-syntax). The same principles that you can use here for plugins also apply to Morpheus packages (https://docs.morpheusdata.com/en/latest/administration/integrations/packages.html).

For plugins, you will need to create a folder called **scribe** in your your plugin's resources folder (`src/main/resources/scribe`). Then add scribe files to that folder. We recommend descriptive names like *Ubuntu22Layouts.scribe*. The scribe files should be modeled off of plugin database model classes (https://developer.morpheusdata.com/api/com/morpheusdata/model/package-summary.html). The resource type should be the model name in dash-case and the resource should have a unique identifier (the `code` field for most models). All new scribe files and updates to existing those files will be processed when your plugin is installed or reloaded.

Resources can reference other resources in the same .scribe file, other resources from different .scribe files, resources from other plugins, and even existing resources in Morpheus. The references are made by specifying the resource type and a unique identifier, often the code. Just make sure that if your plugin expects resources from other plugins or packages, those plugins or packages are loaded first.

Examples

This is an option type (https://developer.morpheusdata.com/api/com/morpheusdata/model/OptionType.html). Note that the `optionSource` field is a reference to a DatasetProvider (https://developer.morpheusdata.com/docs#dataset-providers).

HCL

```
resource "option-type" "demo-option" {
        name = "Demo Option"
        code = "demo-option"
        fieldName = "demoOption"
        fieldContext = "config"
        fieldLabel = "Demo Option"
        type = "select"
        displayOrder = 10
        required = true
        optionSource = "demoOptionSource"
}
```

This is an instance type (https://developer.morpheusdata.com/api/com/morpheusdata/model/InstanceType.html) that references the option type above. They could be in the same .scribe file, or different files. However, if they come from different plugins, make sure that the plugin that provides the option type is loaded first.

```HCL
resource "instance-type" "demo-instance" {
        name = "Demo Instance"
        code = "demo-instance"
        description = "Spin up any VM on our Demo infrastructure."
        environmentPrefix = "DEMO"
        category = "cloud"
        active = true
        enabled = true
        versions = ["1.0"]
        optionTypes = [
                option-type.demo-option
        ]
        provisionTypeDefault = true
        pluginIconPath = "demo.svg"
        pluginIconHidpiPath= "demo.svg"
        pluginIconDarkPath = "demo-dark.svg"
        pluginIconDarkHidpiPath = "demo-dark.svg"
}
```

This is an example instance type layout (https://developer.morpheusdata.com/api/com/morpheusdata/model/InstanceTypeLayout.html) that references the built-in Morpheus Ubuntu instance type. The `provisionType` type here references a ProvisionProvider from a Cloud Plugin (https://developer.morpheusdata.com/docs#cloud-provider-plugins).

```HCL
resource "instance-type-layout" "demo-ubuntu-22" {
        code = "demo-ubuntu-22"
        name = "Demo VM"
        sortOrder = 22
        instanceVersion = "22"
        description = "This will provision a single vm"
        instanceType {
                code = "ubuntu"
        }
        serverCount = 1
        hasAutoScale = true
        portCount = 1
        serverType = "vm"
        enabled = true
        creatable = true
        supportsConvertToManaged = true
        provisionType = "demo-provision-provider"
}
```

Examples

Approvals plugin

Integrate Morpheus with your own ITSM solution. See a full example (https://github.com/gomorpheus/morpheus-plugin-samples/tree/main/morpheus-approvals-plugin).

Setup

This plugin will enable you to create configuration for several aspects of Approvals within Morpheus.

Integration

A new Integration Type will be created when this plugin is installed. You are able to customize the `OptionType` for the new Integration using the `ApprovalProvider.integrationOptionTypes` method. These `OptionType` will be visible when creating the new Integration in the Morpheus UI (Administration → Integrations).

Policies

Policies (Administration → Policies in the Morpheus UI) define the conditions in which approval is required for provisioning. Custom `OptionType` can be defined for Policy creation by implementing the `ApprovalProvider.policyOptionTypes` method.

Create Approval

`ApprovalProvider.createApprovalRequest` is called after a Provision Request is created. Here is where you can send the request to your ITSM. Each `Request` will have one or more `RequestReference` for each resource associated with the provision request.

> **IMPORTANT** It is important that you specify an `externalId` in the `RequestResponse` and each `RequestReference` so that Morpheus can track the approval status.

The `integrationOptionTypes` you specified are available in the method argument `Policy.configMap`

```groovy
String itsmEndpoint = accountIntegration.configMap?.cm?.plugin?."itsm-endpoint"
```

and the `policyOptionTypes` you specified are available in the method argument `AccountIntegration.configMap.`

```groovy
String myPolicyConfigValue = policy.configMap?."my-policy-config-option"
```

Monitor Approval

At a regular interval, Morpheus checks for Request approvals. In the `ApprovalProvider.monitor` method define your logic for retrieving a list of approval requests in your ITSM solution.

Approval `RequestReference` should be returned with one of the following `ApprovalStatus`:

- `requesting`
- `requested`
- `error`

- `approved`
- `rejected`
- `cancelled`

Integration Logo

A custom logo can be used in the Morpheus UI by placing an image at `src/assets/images/{plugin-code}.png`. Recommended file size is 180 x 60 px.

Backups Plugin

Backups are a big part of self-service workload management. Allowing a customer to back up their application and especially restore it on the fly are critical to creating a complete self-service solution. This is why Morpheus integrates with several best-in-class backup solutions like Veeam, Rubrik, Commvault, and more. Morpheus even provides its own basic backup providers for those customers still looking for a final solution.

Plugin Setup

First we need to register the backup provider in the plugin

GROOVY

```
class MyPlugin extends Plugin {

        @Override
        String getCode() {
                return 'my-plugin'
        }

        @Override
        String getName() {
                return 'My Plugin'
        }

        @Override
        void initialize() {
                MyPluginBackupProvider backupProvider = new MyPluginBackupProvider(this, morpheus)
                registerProvider(backupProvider)
        }
```

Creating a BackupPlugin involves registration of 2 types of providers: Backup Provider (https://developer.morpheusdata.com/api/com/morpheusdata/core/backup/BackupProvider.html) and Backup Type Provider (https://developer.morpheusdata.com/api/com/morpheusdata/core/backup/BackupTypeProvider.html).

Backup Provider

The primary entry point into the plugin is the Backup Provider. The backup provider will handle all the high level operations within morpheus including creating and syncing the integration.

The example below is Backup provider is a single Backup Provider with multiple Backup Type Providers.

```groovy
class MyPluginBackupProvider extends AbstractBackupProvider {

        MyPluginBackupProvider(Plugin plugin, MorpheusContext morpheusContext) {
                super(plugin, morpheusContext)

                MyVmwareBackupProvider vmwareBackupProvider = new MyVmwareBackupProvider(plugin, morpheus)
                plugin.registerProvider(vmwareBackupProvider)
                addScopedProvider(vmwareBackupProvider, "vmware", null)

                MyHypervBackupProvider hypervBackupProvider = new MyHypervBackupProvider(plugin, morpheus)
                plugin.registerProvider(hypervBackupProvider)
                addScopedProvider(hypervBackupProvider, "hyperv", null)
        }

        @Override
        String getCode() {
                return 'my-backup-provider'
        }

        @Override
        String getName() {
                return 'My Backup Provider'
        }

        @Override
        Icon getIcon() {
                return new Icon(path:"icon.svg", darkPath: "icon-dark.svg")
        }

        ...
}
```

NOTE: The **AbstractBackupProvider** is convenient to avoid implementing standard methods and settings defined in the **BackupProvider** interface that are not used as often.

A Backup Type Provider **MyVmwareBackupProvider** is created and scoped by a provision type code. Adding a scoped provider will associate the provider to the provision type defined by the scoped provider. In the example a backup provider is defined for provisioning VMware instances.

Backup Type Provider

All operations regarding the instance of a backup, including the creation and restoring of a backup, are handled by the **BackupTypeProvider.** A Backup Provider also can have many Backup Type Providers. For example some backup solutions can back up VMs on many cloud types and they each may have different relevant APIs.

```groovy
class MyVmwareBackupProvider extends AbstractBackupTypeProvider {

    MyVmwareBackupProvider(Plugin plugin, MorpheusContext context) {
        super(plugin, context)
    }

    @Override
    String getCode() {
        return "my-vmware-backup-provider"
    }

    @Override
    String getName() {
        return "MY VMware backup provider"
    }

    @Override
    Collection<OptionType> getOptionTypes() {
        return new ArrayList()
    }

    @Override
    BackupExecutionProvider getExecutionProvider() {
        return new SnapshotExecutionProvider()
    }

    @Override
    BackupRestoreProvider getRestoreProvider() {
        return new SnapshotRestoreProvider()
    }

    ...
}
```

The implementation from this point is flexible. The developer can choose to implement the execute and restore functionality directly in the backup type provider or in separate providers. The execute and restore providers are simply a way to organize the provider for clarity. There are various service interfaces provided for implementing the backup and restore behavior: **BackupExecutionProvider**, **BackupRestoreProvider.**

See the Rubrik Plugin (https://github.com/gomorpheus/morpheus-rubrik-plugin) for a full example implementation including execution and restore providers.

Syncing Provider Data

Like most other integrations, a periodic refresh method is called to sync in any necessary data the integration might need. It is recommended to use a SyncTask (https://developer.morpheusdata.com/api/com/morpheusdata/core/util/SyncTask.html) in these refresh methods which are optimized to handle blocking vs non-blocking thread scheduling.

GROOVY

```groovy
@Slf4j
class MyPluginBackupProvider extends AbstractBackupProvider {

        /... .../

        @Override
        ServiceResponse refresh(BackupProvider backupProvider) {
                ServiceResponse rtn = ServiceResponse.prepare()
                try {
                        new BackupSyncTask().execute()
                } catch(Exception e) {
                        log.error("error refreshing backup provider {}::{}: {}", plugin.name, this.name, e)
                }
                return rtn
        }

        /... .../

}
```

Custom Views

A backup provider can supply custom UI tabs or override the entire backup integration detail view. Override the `renderTemplate()` method to override the integration view. When using the default integration view additional tabs can be added by extending the `AbstractBackupIntegrationTabProvider` class.

The example below shows how to add a custom tab to the default backup provider detail view.

GROOVY

```groovy
class MyBackupTabProvider extends AbstractBackupIntegrationTabProvider {

        /... .../

        @Override
        HTMLResponse renderTemplate(BackupProvider backupProvider) {
                ViewModel<BackupProvider> model = new ViewModel<>()
                model.object = backupProvider

                return getRenderer().renderTemplate("hbs/myTabView", model)
        }

        @Override
        Boolean show(BackupProvider backupProvider, User user, Account account) {
                // only show this tab for providers that match this plugin's backup provider
                return backupProvider.type.code == MyBackupProvider.PROVIDER_CODE
        }

}
```

Override the enter view by implementing the renter template method in your backup provider class:

```groovy
class MyBackupProvider extends AbstractBackupProvider {

        private HandlebarsRenderer renderer

        /... .../

        @Override
        HTMLResponse renderTemplate(com.morpheusdata.model.BackupProvider backupProvider) {
                ViewModel<com.morpheusdata.model.BackupProvider> model = new ViewModel<>()
                model.object = backupProvider

                return getRenderer().renderTemplate("hbs/myBackupProviderView", model)
        }

        // we need to implement a renderer to use handlebars
        Renderer<?> getRenderer() {
                if(renderer == null) {
                        renderer = new HandlebarsRenderer("renderer", getPlugin().getClassLoader())
                        renderer.registerAssetHelper(getPlugin().getName())
                        renderer.registerNonceHelper(getMorpheus().getWebRequest())
                        renderer.registerI18nHelper(getPlugin(),getMorpheus())
                }
                return renderer
        }
}
```

Morpheus Backup Provider

A full backup provider implementation may not be required in many cases. The Morpheus Backup Provider can be used to handle all the high level operations. The example below would allow Morpheus to manage the backup job and delegate the backup execution and restore to back to the plugin's Backup Type Providers.

```groovy
class MyBackupProvider extends MorpheusBackupProvider {

        MyBackupProvider(Plugin plugin, MorpheusContext context) {
                super(plugin, context)

                MySnapshotBackupProvider mySnapshotBackupProvider = new MySnapshotBackupProvider(plugin, morpheus)
                plugin.registerProvider(mySnapshotBackupProvider)
                addScopedProvider(mySnapshotBackupProvider, "vmware", null)
        }

}
```

See the DigitalOcean Plugin (https://github.com/gomorpheus/morpheus-digital-ocean-plugin) for a full example implementation of plugin that utilizes the Morpheus Backup Provider.

Catalog Layouts Plugin

It is becoming popular for some enterprises and managed service providers to expose simpler options when it comes to provisioning workloads. These could be used to target employees who are not highly technical or to further restrict what someone is allowed to order. Not only can they provision workloads like vms, and containers, but also execute operational tasks created by the administrator. Sometimes, it is necessary to further customize how a catalog detail page looks. There may be special ways of displaying information, or even the order form components need some advanced customization.

This plugin exposes the ability to control everything from the HTML used to render the catalog item, to the javascript that controls the form options. By default, we use a server-side handlebars template renderer however this can be completely customized if so desired.

Setup

Given the advanced nature of this plugin, it may be best to start with the sample plugin (https://github.com/gomorpheus/morpheus-plugin-samples/tree/main/morpheus-standard-catalog-layout-plugin) provided in the plugin sample repository (https://github.com/gomorpheus/morpheus-plugin-samples). This plugin replicates the embedded layout functionality so acts as a great starting point. It even includes the javascript used for rendering the option types within it.

> **TIP** Reference the Sample Catalog Layout Plugin before making your own.

The core of the plugin starts with the `CatalogItemLayoutProvider` which extends the common `UIProvider`. Most of the UI related plugin types have some commonalities. The primary difference is the command line arguments sent to the `render()` method.

GROOVY

```
/**
 * Example TabProvider
 */
class StandardCatalogLayoutProvider extends AbstractCatalogItemLayoutProvider {
        Plugin plugin
        MorpheusContext morpheus

        String code = 'catalog-item-standard'
        String name = 'Standard Catalog Layout'

        StandardCatalogLayoutProvider(Plugin plugin, MorpheusContext context) {
                this.plugin = plugin
                this.morpheus = context
        }

    /**
     * Demonstrates building a TaskConfig to get details about the Server and renders the html from the specified
template.
     * @param server details of a ComputeServer
     * @return
     */
        @Override
    HTMLResponse renderTemplate(CatalogItemType catalogItemType, User user) {
        ViewModel<CatalogItemType> model = new ViewModel<>()
        model.object = catalogItemType
        getRenderer().renderTemplate("hbs/standardCatalogItem", model)
    }

}
```

The render method allows the `CatalogItemType` model to be passed into a handlebars view for rendering.

The handlebars template, in this case, takes over the rendering of the entire page below the main navigation. It allows inclusion of external assets as well as assets included in the project via `asset-pipeline`.

```HTML
<script src="{{asset "/form_manager.js"}}" ></script>
<link rel="stylesheet" type="text/css" href="{{asset "/styles.css"}}">
<script src="{{asset "/templates/plugin-configurable-option.js"}}" ></script>
<div class="page-content">
    <div class="catalog-item-details">
        <div class="item-type-header">
            <div class="item-type-image">
                <img class="item-header-img" src="{{catalogTypeImage}}" title="{{name}}"
onError="loadImage(this);"/>
            </div>
            <h1 class="ellipsize" title="{{name}}">{{name}}</h1>
            <div class="desc">{{description}}</div>
        </div>
        <div class="catalog-item-body break-container-sm">
            {{#hasWiki}}
                <div class="catalog-item-content wiki-content">
                    {{wiki}}
                </div>
            {{/hasWiki}}
            <div class="catalog-item-configuration {{#hasWiki}}with-item-content{{/hasWiki}}">
                {{#orderForm}}
                    <div class="actions text-right">

                    </div>
                {{/orderForm}}
            </div>
        </div>
    </div>
</div>
```

NOTE: A couple helper methods are registered such as the `orderForm` block which injects the order form data into the html render.

> **TIP** Pay special attention to the included javascript files used for rendering options. More often than not, one would want to copy these for use in a custom layout.

Consuming the Plugin

Once a Catalog Layout plugin is compiled and loaded into a Morpheus environment, the layout is automatically made available globally for use when creating a service catalog item in the `Blueprints` section. Simply edit the catalog item and a new dropdown showing available layouts should be available to choose.

Cloud Provider Plugins

The Cloud provider plugin interfaces are among the more complicated plugin types to implement within Morpheus, but when implemented successfully it provides a very powerful method for creating custom clouds or even updating existing cloud functionality. There are two primary concepts that must first be discussed when developing a cloud plugin.

A Cloud plugin typically defines a cloud type (you may see this as `zoneType` in api for legacy compatibility.) as well as at least one `ProvisionProvider`. A Provision Provider (also seen in api as `provisionType`) defines how a resource is provisioned within a cloud. A cloud could offer many provisioning types. For example, Amazon offers both `EC2` as well as `RDS`.

There are several other provider implementations that are needed on more advanced cloud implementations and some are not yet built out. This includes `NetworkProvider` and `BackupProvider` implementations as well as a few more.

Setup

Before Getting Started, It is recommended to look at the digital ocean sample plugin just to get some bearings. To get started you firstly will create a new class that implements `com.morpheusdata.core.CloudProvider`. The `CloudProvider` requires implementation of several methods including the code required to sync existing workloads from the cloud. All sync related code normally lives in here.

On a cloud implementation there are 2 scheduled jobs for refreshing. There is firstly a 5 minute sync job (typically) that syncs state changes on a periodic basis. Then secondly a daily job that runs at Midnight UTC to do larger syncs like price data or things that may change less frequently. Refer to implementations of the `refresh()` method as well as the `refreshDaily()` methods.

Finally one must also provide the option types inputs for configuring the add cloud wizard as well as the `ComputeServerType` objects available to this zone type. There should be multiple based often on platform and management state.

Defining Credentials

During the `refresh()` call (and others) the `CloudProvider` implementation will need to reach out to underlying cloud using credentials. There are two ways to define and store the required credentials.

Using OptionTypes

The simplest method is to define `OptionType` fields (like username and password) and return them on implementation of `CloudProvider.getOptionTypes()`. These will then be displayed on the Cloud configuration UI. The values can then be obtained during sync operations via the Cloud's `getConfigMap` or directly on the object itself (i.e. `serviceUsername`) depending on how the `OptionType` was defined.

Using Morpheus Credentials

A more flexible option is to use Morpheus' built-in Credentials support. With this option, Credentials can be stored securely in Morpheus and utilized in various locations. In order to user this method, a few specific `OptionType` objects need to be defined. (Refer to `VmwareCloudProvider` for an example implementation)

1. An `OptionType` needs to be defined to represent the selection of the Credential type. The following properties must be configured on the `OptionType`: `inputType=OptionType.InputType.CREDENTIAL`, `fieldContext=credential`, `fieldName=type`, `optionSource=credentials`. In addition, the `config` for the `OptionType` should be something like `'{"credentialTypes":["username-password"]}'`. Where the array of types may be one or more of `username-password`, `username-password-keypair`, `username-keypair`, `access-key-secret`, `client-id-secret`, `username-api-key`, `email-private-key`, `tenant-username-keypair`, `oauth2`, `api-key`. These represent the preconfigured credential types in Morpheus.
2. `OptionType` objects need to be defined to represent the 'local' auth values. For example, username and password would need their own `OptionType`. For these 'local' types, their `localCredential` value must be `true`
3. Any `OptionTypes` that should be reloaded when the Credential input changes should include `credential-type` in their `dependsOn` value. This will trigger the `OptionType` `optionSource` function to be called when the Credentials change.

To load the Credential information that may be set on a Cloud, the `MorpheusCloudService` can be used.

To load the Credential information from within `OptionSourceService` implementations (which may be called during Cloud configuration), `MorpheusAccountCredentialService` may be used to load Credential information from the passed in form options. See `VmwareOptionSourceService` for an example.

Datastore Type Providers

A Datastore Type Provider allows the plugin developer to implement custom Datastore types for use with various provision providers. **(Currently only VME/MVM is supported)**. This could be for use with third party storage arrays or in the event some other type of Software Defined Storage (SDS) is desired to be used with the VME platform.

What is the difference between a Datastore Type Provider and a Storage Provider?

Storage Providers are typically used for managing storage outside the general provisioning workload pipeline. This allows for volume and share management within a datastore that can be tied into automation, but does not directly affect provisioning operations. Where as, a Datastore, is a storage target for a workload to run on. This provides additional methods that give context that can leverage the Storage Server (defined by the Storage Provider).

Getting Started

Before getting started, it is recommended that a `StorageProvider` also be created along with a `DatastoreTypeProvider` plugin. This is because they often go together. For example, an Alletra Storage Array can be registered by the user as a Storage Integration via the `StorageProvider` and then the `DatastoreTypeProvider` can be used to define the various types of Datastore objects that can be created using this storage array (i.e. iSCSI LUN per vDISK).

Another common provider type that goes along with this is the `BackupProvider`. Any type of custom backup implementation should leverage that provider along with this.

NOTE: Pay special attention to the `Facet` options on the interface. These provide additional functionality when they are implemented on the provider class.

Create a new class that implements `com.morpheusdata.core.providers.DatastoreTypeProvider` and implement the required methods as well as the `StorageProvider.

```groovy
class MyCustomDatastoreTypeProvider implements DatastoreTypeProvider, DatastoreTypeProvider.MvmProvisionFacet,
DatastoreTypeProvider.SnapshotFacet.SnapshotServerFacet {

        @Override
        String getStorageProviderCode() {
                return 'my-storage-provider-code'
        }
// Implement required methods here for all 3 interfaces

}

class MyCustomStorageProvider implements StorageProvider {
        @Override
        String getCode() {
                return 'my-storage-provider-code'
        }
}
```

Your favorite IDE (such as IntelliJ IDEA) should be able to auto-generate the required methods for you. Or, you can use the `Getting Started` plugin generator on the developer portal to create a new Datastore Type Provider.

Facets

In the above example, two additional facets were injected into the class. These control implementation of Snapshots as well as the ability to inject behaviors specific to VME/MVM Provisioning. For example, in some scenarios it may be necessary to prepare the host for a vm move or even an initial provision of a vm. These allow for injection points for this.

The `MVMProvisionFacet` also contains a means to override/customize the libvirt Domain XML block device specification. This can be important if using a custom storage array implementation that may use raw BLOCK or iSCSI vs QCOW2 formats.

Host Interactions

When utilizing the `DatastoreTypeProvider` for VME/MVM it is often necessary to execute automation directly on the hypervisor. It is recommended that this be performed with the `morpheusContext.executeCommandOnServer` method. This will automatically ensure the command is executed through the right medium (such as through an Edge Distributed Worker and through the agent with proper ssh fallback).

The host of a specific vm/workload is normally found by grabbing the `ComputeServer.getParentServer()` method off the workload server representation.

Snapshots

Snapshots are a common feature of most storage arrays and SDS solutions. The `SnapshotFacet` provides a means to implement as well as the `SnapshotServerFacet` which is more commonly used. The key difference is one implementation focuses strictly on per volume snapshots, and the other focuses on snapshotting all volumes within a server at once. This is useful for things like backup operations, image export, or just general state.

Generic Integration Plugins

Generic Integration Plugins enable the registration of a "Generic" `AccountIntegrationType`. A Generic Integration plugin is designed to add additional functionality to existing providers and extend the capability of Morpheus where existing providers are not available.

For example, a Jenkins plugin implementing a Generic Integration Provider could store credentials for Jenkins task types, allowing users to avoid entering credentials repeatedly when creating new tasks. Additionally, the plugin could periodically sync data, such as a list of projects, used the `DatasetProvider` dropdowns in the Jenkins task types.

A Generic Integration example can be found in the sample plugin repository (https://github.com/gomorpheus/morpheus-plugin-samples/tree/main/morpheus-generic-integration-plugin) on GitHub.

Network Provider Plugins

The Network Provider Plugin interface requires either an associated `GenericIntegrationProvider` or `CloudProvider` provider in the plugin to provide the account integration or cloud backing for the network server as well as sharing icons and other resources. Make sure to set the appropriate code in the `NetworkProvider` implementation.

IPAM/DNS Plugins

The IPAM and DNS Provider Plugin interfaces provide an easy means to create direct orchestration for IP Address allocation/release as well as DNS Name server registrations. An IPAM Provider is often capable of implementing both interfaces as they often provide both services at once. It is also possible to independently register a `DNSProvider` only.

Both the DNS and IPAM Provider plugins typically consist of just 3 parts. First is defining the provider information such as the configuration options for adding the integration as well as pool types it offers. Secondly periodically syncing state data into morpheus back from the remote integration endpoints. This could include just syncing in available pools or zones, all the way to syncing in all IP host records or DNS zone records. This is up to your implementation.

Setup

Before Getting Started, It is recommended to look at the infoblox plugin just to get some bearings. To get started you firstly will create a new class that implements `com.morpheusdata.core.IPAMProvider` and `com.morpheusdata.core.DNSProvider`. The Providers requires implementation of several methods including the code required to sync existing records. All sync related code normally lives in here.

Both providers also require implementing CRUD based methods for creating host records, deleting host records, and creating zone records and deleting zone records. It is important to note that the host record object allows a user to directly enter an ip address to be requested for allocation or, if none is provided, it should be assumed the next available IP should be acquired. Host records are also special in that there are additional options for simultaneously creating DNS records such as A and PTR records. The additional complexity of tieing these pieces into the automated provisioning of workloads is hidden and taken care of by the Morpheus orchestrator.

Reports Plugin

Create custom report types for users to consume within Morpheus. Customize the behavior of how the report data is assembled and generated as well as the way it is rendered/displayed to the user. See a full example (https://github.com/gomorpheus/morpheus-plugin-samples/tree/main/morpheus-reports-plugin).

Setup

Creating a report is just a matter of registering a new implementation of a `ReportProvider`. If using standard handlebars rendering similar to UI Extensions, simply extend the `com.morpheusdata.core.AbstractReportProvider`. Before creating a report a few concepts should be made known.

There are a few model objects of importance. Firstly, the `ReportProvider` implementation always generates a `ReportType` for reference and display when the user is browsing a report ro run.

After a report is run a `ReportResult` is generated. This represents the information of who created the report as well as any submitted filters / report options related to the report. When creating a process method the results of the run should be stored as `ReportResultRow` objects. These have a `displayOrder` and `section`. This allows one to store header data as well as line item data for rendering and csv export. These rows are generated using rxjava asynchronous flows in the `process` method. Example Here:

GROOVY

```
void process(ReportResult reportResult) {
  morpheus.report.updateReportResultStatus(reportResult,ReportResult.Status.generating).blockingGet()
  Long displayOrder = 0
  List<GroovyRowResult> results = []
  withDbConnection { Connection dbConnection ->
    if(reportResult.configMap?.phrase) {
      String phraseMatch = "${reportResult.configMap?.phrase}%"
      results = new Sql(dbConnection).rows("SELECT id,name,status from instance WHERE name LIKE ${phraseMatch}
order by name asc;")
    } else {
      results = new Sql(dbConnection).rows("SELECT id,name,status from instance order by name asc;")
    }
  }

Observable<GroovyRowResult> observable = Observable.fromIterable(results) as Observable<GroovyRowResult>
  observable.map{ resultRow ->
    Map<String,Object> data = [name: resultRow.name, id: resultRow.id, status: resultRow.status]
    ReportResultRow resultRowRecord = new ReportResultRow(section: ReportResultRow.SECTION_MAIN, displayOrder:
displayOrder++, dataMap: data)
    return resultRowRecord
  }.buffer(50).doOnComplete {
    morpheus.report.updateReportResultStatus(reportResult,ReportResult.Status.ready).blockingGet()
  }.doOnError { Throwable t ->
    morpheus.report.updateReportResultStatus(reportResult,ReportResult.Status.failed).blockingGet()
  }.subscribe {resultRows ->
    morpheus.report.appendResultRows(reportResult,resultRows).blockingGet()
  }
}
```

NOTE: Notice that this process method features the ability to get a read only database connection to the morpheus MySQL Database. This isn't always the best option but is a good fallback option for grabbing data you may not otherwise be able to get. Other data query methods are available on the various `MorpheusContext` subService classes. Expect more of these to be filled out as the plugin ecosystem develops. A good example of this is the `MorpheusAccountInvoiceService` found via the `MorpheusCostingService`. It enables you to query all invoices just as you would from the api.

Custom Filters

It is often the case that a user may want to adjust how a filter runs. Perhaps they want to reduce the result set to a filtered set of data or group by certain properties. For this, the `ReportProvider` provides a `getOptionTypes` method that when implemented allows the developer to specify custom form inputs the user has to select when running the report.

GROOVY

```
@Override
List<OptionType> getOptionTypes() {
  [new OptionType(code: 'status-report-search', name: 'Search', fieldName: 'phrase', fieldContext: 'config',
fieldLabel: 'Search Phrase', displayOrder: 0)]
}
```

It is important to note the `fieldContext` should almost always be set to `config` in this instance.

Rendering

Render is very similar to rendering a tab. The main difference is the payload that is sent for the render is the `ReportResult` representing the particular report run as well as the dataset rows grouped by section.

```groovy
@Override
HTMLResponse renderTemplate(ReportResult reportResult, Map<String, List<ReportResultRow>> reportRowsBySection) {
    ViewModel<String> model = new ViewModel<String>()
    model.object = reportRowsBySection
    getRenderer().renderTemplate("hbs/instanceReport", model)
}
```

TIP When using custom javascript or stylesheets be sure to use the provided `{{nonce}}` helper to inject the appropriate nonce token for the `Content-Security-Policy`.

Task Plugin

Add custom Tasks types to Morpheus. See a full example (https://github.com/gomorpheus/morpheus-plugin-samples/tree/main/morpheus-task-plugin).

Setup

Tasks are useful components of your provisioning workflow. This plugin allows you to create custom Tasks

- Create a new class that implements `com.morpheusdata.core.TaskProvider`
- Create a new class that extends `com.morpheusdata.core.AbstractTaskService`. This service defines methods for task execution in a variety of contexts, described below.

Options

`OptionType` is an easy way to create configuration for your new Task. Simply provide a list of `com.morpheusdata.model.OptionType` to the `TaskProvider.getOptionTypes` method.

```groovy
        @Override
        List<OptionType> getOptionTypes() {
                OptionType optionType = new OptionType(
                                name: 'myTask',
                                code: 'myTaskText',
                                fieldName: 'myTask',
                                optionSource: true,
                                displayOrder: 0,
                                fieldLabel: 'Text to Reverse',
                                required: true,
                                inputType: OptionType.InputType.TEXT
                )
                return [optionType]
        }
```

Task Contexts

A task can be run in one of three contexts:

- None/Local (`executeLocalTask`)
- Remote (`executeRemoteTask`)
- Instance (`executeContainerTask` , `executeContainerTask`)

Task Logo

A custom logo can be used in the Morpheus UI by defining the `Icon` object in the new `TaskProvider.getIcon()` interface method. Before this was simply a hard coded icon referenced by a code name. Both dark mode and light mode icons can be defined.

UI Extensions

Morpheus UI Extension Plugins provide a way to expand the capabilities of the Morpheus UI. Render custom content as a tab on an Instance, or even on a Server. Create a global injection component to the main layout footer for support/chat services. The possibilities are growing with each release and new functionality since 0.8.0 brings a lot. See a full tabs example (https://github.com/gomorpheus/morpheus-plugin-samples/tree/main/morpheus-tab-plugin).

Setup Instance Tabs

Create a new class that extends `com.morpheusdata.core.AbstractInstanceTabProvider` . When the Morpheus UI builds the Instance UI it calls the `renderTemplate` method. Below is a simple example binding the Instance object to the template model.

GROOVY

```
@Override
HTMLResponse renderTemplate(Instance instance) {
    ViewModel<Instance> model = new ViewModel<>()
    model.object = instance
    getRenderer().renderTemplate("hbs/instanceTab", model)
}
```

> **TIP** Use `MorpheusContext.buildInstanceConfig` to get more details about your Instance. See `com.morpheusdata.model.TaskConfig`

Handlebars (https://github.com/jknack/handlebars.java) is the default provided template engine. To override this default, implement the `com.morpheusdata.core.InstanceTabProvider` interface and then write your own `getRenderer()` method.

> **TIP** The `HandlebersRenderer` provides a helper called `{{nonce}}` for injecting into script tags

Setup Server Tabs

Create a new class that extends `com.morpheusdata.core.AbstractServerTabProvider` . When the Morpheus UI builds the Server Details UI, it calls the `renderTemplate` method. Below is a simple example binding the Instance object to the template model.

GROOVY

```
@Override
HTMLResponse renderTemplate(ComputeServer server) {
    ViewModel<ComputeServer> model = new ViewModel<>()
    model.object = server
    getRenderer().renderTemplate("hbs/serverTab", model)
}
```

> **TIP** Use `MorpheusContext.buildComputeServerConfig` to get more details about your Server. See `com.morpheusdata.model.TaskConfig`

Handlebars (https://github.com/jknack/handlebars.java) is the default provided template engine. To override this default, implement the `com.morpheusdata.core.ServerTabProvider` interface and then write your own `getRenderer()` method.

Templating

See the Views section and the documentation for your templating engine for specific syntax.

Security Policies

User Permissions

Before a template is rendered in the UI, the `InstanceTabProvider.show` method is called to determine if the current user can view the custom Instance Tab. For example, you may wish to check that the current `User` has been granted the custom permission defined by your plugin.

GROOVY

```
@Override
Boolean show(Instance instance, User user, Account account) {
    def show = true
    plugin.permissions.each { Permission permission ->
        if(user.permissions[permission.code] != permission.availableAccessTypes.last().toString()){
            show = false
        }
    }
    return show
}
```

Content Security Policy

If your custom UI needs to include external resources such as scripts, stylesheets, or frames, you may need to customize the Morpheus `Content-Security-Policy` Header to allow those elements to be loaded in the browser.

GROOVY

```
@Override
TabContentSecurityPolicy getContentSecurityPolicy() {
    def csp = new TabContentSecurityPolicy()
    csp.scriptSrc = '*.jsdelivr.net'
    csp.frameSrc = '*.digitalocean.com'
    csp.imgSrc = '*.wikimedia.org'
    csp.styleSrc = 'https: *.bootstrapcdn.com'
    csp
}
```

You can also use the per request `nonce` token to set the attribute on the `<script/>` tags you may be injecting:

HANDLEBARS

```
<script src="blah.js" nonce="{{nonce}}"/>
```

Version 1.2.7
Last updated 2025-04-10 20:09:47 UTC