160 Feature Launcher Service Specification

Version 1.0

160.1 Introduction

The *Feature Service Specification* on page 71 defines a model to design and declare Complex Applications and reusable Sub-Components that are composed of multiple bundles, configurations and other metadata. These models are, however, only descriptive and have no standard mechanism for installing them into an OSGi framework.

This specification focuses on turning these Features into a running system, by introducing the Feature Launcher and Feature Runtime. The Feature Launcher takes a Feature definition, obtains a framework instance for it and then starts the Feature in that environment. The Feature Runtime extends this capability to a running system, enabling one or more Features to be installed, updated, and later removed from a running OSGi framework.

The Launcher and Runtime also interact with the Configuration Admin Service, that is, they provide configuration to the system if it is present in the Feature being launched or installed.

160.1.1 Essentials

- *Dynamic* - The Feature Runtime dynamically adds, updates and removes Features in a running system.
- *Parameterizable* - Feature installation may be customised using local parameters if the Feature supports it.
- *Zero code* - The Feature Launcher can launch a framework containing an installed Feature in an implementation independent way without a user writing any code .

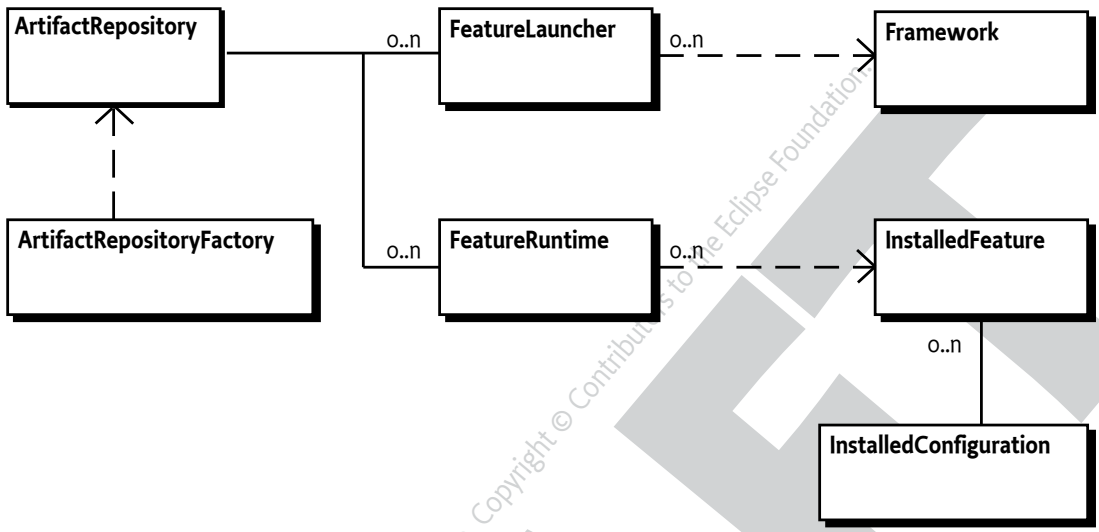
160.1.2 Entities

The following entities are used in this specification:

- *Feature* - A Feature as defined by the *Feature Service Specification* on page 71
- *Artifact Repository* - A means of accessing the installable bytes for bundles in a Feature
- *Feature Launcher* - A Feature Launcher obtains an OSGi Framework instance and installs a Feature into it.
- *Framework* - A running implementation of the OSGi core specification.
- *Launch Properties* - Framework launching properties defined in a Feature.
- *Feature Parameters* - Key value pairs that can be used to customise the installation of a Feature.
- *Configuration* - A configuration for the Configuration Admin service.
- *Feature Runtime* - A Feature Runtime is an OSGi service capable of installing Features into the running OSGi framework, removing installed Features from the OSGi framework, and updating an installed Feature with a new Feature definition.
- *Installed Feature* - A representation of a Feature installed by the Feature Runtime.
- *Installed Configuration* - A representation of a Configuration installed by the Feature Runtime.

DRAFT
Licensed under the Eclipse Foundation Specification License 1.0 Copyright © Contributors to the Eclipse Foundation.

Figure 160.1 Features Entity overview



Licensed under the Eclipse Foundation Specification License – v1.0. Copyright © Contributors to the Eclipse Foundation.

160.2 Features and Artifact Repositories

OSGi Features exist either as JSON documents, or as runtime objects created by the Feature Service API. The primary purpose of a Feature is to define a list of bundles and configurations that should be installed, however the Feature provides no information about the location of the bundle artifacts. A key challenge with installing a Feature is therefore finding the appropriate artifacts to install.

The ArtifactRepository interface is designed to be implemented by users of the Feature Launcher Service to provide a way for the Feature Launcher Service to find an installable InputStream of bytes for a given bundle artifact using the getArtifact(ID) method. Artifact Repository implementations are free to use any mechanism for locating the bundle artifact data. If no artifact can be found for the supplied ID then the implementation of the Artifact Repository should return null. If the Artifact Repository throws an exception then this must be logged by the Feature Launcher Service and then treated in the same manner as a null return value.

160.2.1 The Artifact Repository Factory

In order to support the *Zero Code* objective of this specification, and to simplify usage for most users, the ArtifactRepositoryFactory provides a factory for commonly used repository types.

160.2.1.1 Obtaining an Artifact Repository Factory

The Artifact Repository Factory is useful both for the Feature Launcher and the Feature Runtime, and as such it must be easy to access both inside and outside an OSGi framework. The Feature Launcher Service implementation must provide an implementation of the Artifact Repository Factory interface. A user of the Artifact Repository Factory service may use the following ways to find an instance.

When outside OSGi:

- Using the Java ServiceLoader API to find instances of org.osgi.service.featurelauncher.repository.ArtifactRepositoryFactory
- From configuration, and then using Class.forName, getConstructor() and newInstance()

- By hard coding the implementation and using the new operator.

When inside an OSGi framework:

- Using the OSGi service registry to find instances of org.osgi.service.featurelauncher.repository.ArtifactRepositoryFactory
- Using the Java ServiceLoader API and the OSGi Service Loader Mediator to find instances of org.osgi.service.featurelauncher.repository.ArtifactRepositoryFactory
- By hard coding the implementation type and using the new operator.

160.2.1.2 Local Repositories

A Local Repository is one that exists on a locally accessible file system. Note that this does not require that the file system is local, and technologies such as NFS or other network file systems would still be considered as Local Repositories. The key aspects of a Local Repository are that:

- The root of the repository can be accessed and resolved as a java.nio.file.Path or file: URI.
- The repository uses [1] *The Maven 2 Repository Layout*

An Artifact Repository representing a Local Repository can be created using the createRepository(Path) method, passing in the path to the root of the repository. A NullPointerException must be thrown if the path is null and an IllegalArgumentException must be thrown if the path does not exist, or represents a file which is not a directory.

An Artifact Repository representing a Local Repository can also be created using the createRepository(URI,Map) method, passing a URI using the file scheme which points to the root of the repository. A NullPointerException must be thrown if the URI is null and an IllegalArgumentException must be thrown if the path does not exist, or represents a file which is not a directory.

Once created this Artifact Repository will search the supplied repository for any requested artifact data. Implementations are free to optimise checks using repository metadata.

160.2.1.3 Remote Repositories

A Remote Repository is one that exists with an accessible http or https endpoint for retrieving artifact data. Note that this does not require that the repository is on a remote machine, only that the means of accessing data is via HTTP requests. The key aspects of a Remote Repository are that:

- The root of the repository can be accessed and resolved as a http or https URI
- The repository uses [1] *The Maven 2 Repository Layout*

An Artifact Repository representing a Remote Repository can be created using the createRepository(URI,Map) method, passing in the uri to the root of the repository. A NullPointerException must be thrown if the uri is null and an IllegalArgumentException must be thrown if the uri does not use the http or https scheme.

In addition to the repository URI the user may pass configuration properties in a Map. Implementations may support custom configuration properties, but those properties should use Reverse Domain Name keys. Keys not using the reverse DNS naming scheme are reserved for OSGi use. Implementations must ignore any configuration property keys that they do not recognise. All implementations must support the following properties:

- REMOTE_ARTIFACT_REPOSITORY_NAME - The name for this repository
- REMOTE_ARTIFACT_REPOSITORY_USER - The user name to use for authenticating with this repository
- REMOTE_ARTIFACT_REPOSITORY_PASSWORD - The password to use for authenticating with this repository
- REMOTE_ARTIFACT_REPOSITORY_BEARER_TOKEN - A bearer token to use when authenticating with this repository

Licensed under the Eclipse Foundation Specification License – v1.0. Copyright © Contributors to the Eclipse Foundation.
DRAFT

- REMOTE_ARTIFACT_REPOSITORY_SNAPSHOTS_ENABLED - A Boolean indicating that SNAPSHOT versions are supported. Defaults to true
- REMOTE_ARTIFACT_REPOSITORY_RELEASES_ENABLED - A Boolean indicating that release versions are supported. Defaults to true
- REMOTE_ARTIFACT_REPOSITORY_TRUST_STORE - A trust store to use when validating a server certificate. May be a file system path or a data URI as defined by [2] *The Data URI scheme*.
- REMOTE_ARTIFACT_REPOSITORY_TRUST_STORE_FORMAT - The format of the trust store to use when validating a server certificate.
- REMOTE_ARTIFACT_REPOSITORY_TRUST_STORE_PASSWORD - The password to use when validating the trust store integrity.

Once created this Artifact Repository will search the supplied repository for any requested artifact data. Implementations are free to optimise checks using repository metadata.

160.3 Common themes

This specification includes support for bootstrapping an OSGi runtime, for ongoing management of an OSGi runtime, and for merging features. There are many concepts that apply across more than one of these scenarios, and so they are described here.

160.3.1 Overriding Feature variables

Some Feature definitions include variables which can be used to customise their deployment. These variables are intended to be set at the point where a Feature is installed, and may contain default values. To enable these variables to be overridden there are overloaded versions of methods which permit a Map of variables to be provided. The keys in this map must be strings and the values must be one of the types permitted by the *Feature Service Specification* on page 71

If a Feature declares a variable with no default value then this variable *must* be provided. If no value is provided then the method must fail to launch by throwing a LaunchException

160.3.2 Setting the bundle start levels

An OSGi framework contains a number of bundles which collaborate to produce a functioning application. There are times when some bundles require the system to have reached a certain state before they can be started. To address this use case the OSGi framework has the concept of *start levels* as described in the Start Level API Specification chapter of *OSGi Core Release 8.*.

Setting the initial start level for the OSGi framework when bootstrapping can easily be achieved using the framework launch property org.osgi.framework.startlevel.beginning as defined by the OSGi core specification.

Controlling the start levels assigned to the bundles in a feature is managed through the use of Feature Bundle metadata. Specifically the Feature Launcher will look for a Feature Bundle metadata property named BUNDLE_START_LEVEL_METADATA which is of type integer and has a value between 1 and 2147483647 inclusive. If the property does not exist then the default start level will be used. If the property does exist and is not a suitable integer then launching must fail with a LaunchException.

Setting the default start level for the bundles, and the minimum start level required for an installed Feature is accomplished by using a Feature Extension named BUNDLE_START_LEVELS with Type JSON. The JSON contained in this extension is used to configure the default start level for the bundles, and the target start level for the framework. The schema of this JSON is as follows: ### Add Schema in build

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
```

DRAFT
Licensed under the Eclipse Foundation Specification License – v1.0. Copyright © Contributors to the Eclipse Foundation.

```
  "$id": "http://www.osgi.org/json.schema/featurelauncher/v1.0.0/bundle-start-levels.json",
  "title": "bundle-start-levels",
  "description": "The definition of the bundle-start-levels feature extension",
   "type": "object",
   "properties": {
      "version": {
        "description": "The version of the Feature Launcher extension",
        "const": "1.0.0"
      },
      "defaultStartLevel": {
         "description": "The default start level for bundles in the feature",
         "type": "integer",
         "minimum": 1,
         "maximum": 2147483647
      },
      "minimumStartLevel": {
         "description": "The minimum required start level for the framework after feature installation",
         "type": "integer",
         "minimum": 1,
         "maximum": 2147483647
      }
   },
   "required": [ "version", "defaultStartLevel", "minimumStartLevel" ]
}
```

Setting the default start level for bundles installed by the framework is achieved using the defaultStartLevel property of the JSON extension. This must be an integer greater than zero and less than Integer.MAX_INT, or the special marker value null. A null value is used to indicate that the default start level for newly installed bundles is the current framework start level, or 1 if the current framework start level is 0. If the value is not valid then a LaunchException must be thrown when attempting to use the feature.

The minimum final start level for the OSGi framework required by the feature can be set using the minimumStartLevel property. of the JSON extension. This must be an integer greater than zero and less than Integer.MAX_INT. If the value is not valid then a LaunchException must be thrown when attempting to use the feature. This property sets the minimum start level that the OSGi framework must use to complete the installation of a Feature.

Finally the version property defines the version of the extension schema being used. This can be used by the implementation to determine whether the Feature is targeting a newer version of the specification. If the version is not understood by the implementation then a LaunchException must be thrown when attempting to use the feature.

160.3.3 Feature Decoration

Feature Decoration is a process by which features can be pre-processed before they are installed or updated. This gives users an opportunity to modify the feature, accept it as is, or block the operation from proceeding. There are two types of decorator:

- *Feature Decorators* - called for all operations. Can re-write the bundles, configurations, variables and extensions present in a feature.
- *Feature Extension Handlers* - called operations where the feature defines the named extension. Can re-write the bundles, configurations and variables present in a feature, but not the extensions.

Both types of decorator may pass through the feature unchanged by returning the feature object passed into them. This will cause the operation to continue as normal. Decorators may also block an operation from proceeding by throwing an AbandonOperationException. This will cause the operation to be immediately halted, and an exception thrown to the caller who requested the operation.

160.3.3.1 Building decorated features

Feature objects are expected to be immutable, and therefore a decorator cannot, and should not, change the feature object that is passed to them. Instead the decorator must create a new feature object which includes the decorated content.

Licensed under the Eclipse Foundation Specification License 1.0. Copyright © Contributors to the Eclipse Foundation.
DRAFT

To enable this both types of decorator are passed two builders, the first of which implements BaseFeatureDecorationBuilder and the second of which implements DecoratorBuilderFactory.

The former builder is similar to a FeatureBuilder but with three important differences:

- The builder is pre-populated with the information from the existing feature, such that immediately calling build() would create a feature with identical content to the original.
- Except where explicitly stated the builder configuration methods *replace* content rather than adding to it
- Only a limited subset of the feature content can be changed.

The latter builder is similar to a BuilderFactory but it cannot create FeatureBuilder instances.

By using these two builders a decorated feature can be configured and created. This decorated feature can then be returned from the decorator. Note that the *only* valid way to create a decorated feature is by using the builder. Any attempt to return a feature object which is not either:

- The original feature object.
- The object returned by build()

is an error and will result in the operation being abandoned.

160.3.3.2 Using Decorators

Decorators may be included using one of the relevant builder methods for a launch or runtime operation:

- withDecorator(FeatureDecorator)
- withExtensionHandler(String,FeatureExtensionHandler)
- withDecorator(FeatureDecorator)
- withExtensionHandler(String,FeatureExtensionHandler)

When registering a FeatureExtensionHandler the name of the extension to be handled must be passed, and cannot be null. This defines the name of the extension that the Feature Extension Handler will be used to process.

If multiple FeatureDecorator instances are registered then they will be called in the order that they were added.

If multiple FeatureExtensionHandler instances are registered for the same extension name then the earlier instances will be discarded. It is not possible to register more than one Feature Extension Handler for a single extension.

160.4 The Feature Launcher

The FeatureLauncher is the main entry point for creating a running OSGi framework containing the bundles and configurations defined in a Feature. As such the Feature Launcher is primarily designed for use outside of an OSGi framework.

To support usage in a non-OSGi environment implementations of the Feature Launcher Service must register the following implementation classes with the Java ServiceLoader API, and any necessary module metadata.

- org.osgi.service.featurelauncher.FeatureLauncher
- org.osgi.service.featurelauncher.repository.ArtifactRepositoryFactory

DRAFT
Licensed under the Eclipse Foundation Specification License – v1.0 Copyright © Contributors to the Eclipse Foundation.

160.4.1 Obtaining and configuring a Feature Launcher

A Feature Launcher Service implementation must provide an implementation of the Feature Launcher interface. A user of the Feature Launcher service may use the following ways to find this class and create an instance:

- Using the Java ServiceLoader API to find instances of org.osgi.service.featurelauncher.FeatureLauncher
- From configuration, and then using Class.forName, getConstructor() and newInstance()
- By hard coding the implementation type and using the new operator.

Once instantiated the Feature Launcher may be supplied with a Feature, either as a Reader providing access to the JSON text of a Feature document or a parsed Feature to create a FeatureLauncher.LaunchBuilder. The Launch Builder can be configured in a fluent manner using the withConfiguration(Map), withVariables(Map), withFrameworkProperties(Map) and withRepository(ArtifactRepository) methods. Configuration properties for the Feature Launcher are implementation specific, and any unrecognised property names should be ignored. Artifact Repository instances may be created by the user using as described in *The Artifact Repository Factory* on page 94, or using custom implementations.

160.4.1.1 Thread Safety

Instances of the Feature Launcher and Launch Builder are not required to be Thread Safe, and should not be shared between threads. Changing the configuration of a Launch Builder instance only affects that instance, and not any other instances that exist.

160.4.2 Using a Feature Launcher

Once a configured Launch Builder instance has been created the launchFramework() method can be used to launch an OSGi framework containing the supplied Feature. The Feature Launcher will then return a running Framework instance representing the launched OSGi framework and the Feature that it contains. If an error occurs creating the framework, or locating and installing any of the feature bundles, then a LaunchException must be thrown.

Once the caller has received their framework instance they may carry on with other work, or they may wait for the OSGi framework to stop using the waitForStop() method.

160.4.2.1 Providing Framework Launch Properties

Framework launch properties are key value pairs which are passed to the OSGi framework as it is created. They can control many behaviours, including operations which happen before the framework starts, meaning that is not always possible to set them *after* startup.

Feature definitions that require particular framework launch properties can define them using a Feature Extension named FRAMEWORK_LAUNCHING_PROPERTIES. The Type of this Feature Extension must be TEXT, where each entry is in the form key=value All implementations of the Feature Launcher must support this extension, and use it to populate the Framework Launch Properties.

In addition to Framework Launch properties defined inside the Feature, users of the Feature Launcher can add and override Framework Launch Properties using one of the withFrameworkProperties method that permits a Map of framework properties to be provided. Any key value pairs defined in this map must take precedence over those defined in the Feature. A key with a null value must result in the removal of a key value pair if it is defined in the Feature.

160.4.2.2 Selecting a framework implementation

When defining a feature it is not always possible to be framework independent. Sometimes specific framework APIs, or licensing restrictions, will require that a particular implementation is used. In this case a Feature Extension named LAUNCH_FRAMEWORK with Type ARTIFACTS can be used to list one or more artifacts representing OSGi framework implementations.

Licensed under the Eclipse Foundation Specification License - v1.0. Copyright (c) OSGi Alliance and the Eclipse Foundation.
DRAFT

The list of artifacts is treated as a preference order, with the first listed artifact being used if available, and so on, until a framework is found. If a listed artifact is not an OSGi framework implementation then the Feature Launcher must log a warning and continue on to the next artifact in the list. If the Kind of the feature is MANDATORY and none of the listed artifacts are available then launching must fail with a LaunchException.

The Feature Launcher implementation may identify that an artifact is an OSGi framework implementation in any way that it chooses, however it must recognise framework implementations that provide the Framework Launch API using the service loader pattern, as described in the Launching and Controlling a Framework section of *OSGi Core Release 8*.

160.4.2.3 A simple example

The following code snippet demonstrates a simple example of using the Feature Launcher to start an OSGi framework containing one or more bundles.

```
// Load the Feature Launcher
ServiceLoader<FeatureLauncher> sl = ServiceLoader.load(FeatureLauncher.class);
FeatureLauncher launcher = sl.iterator().next();

// Set up a repository
ArtifactRepository localRepo = launcher.createRepository(Paths.get("bundles"));

// Launch the framework
Framework fw = launcher
        .launch(Files.newBufferedReader(Paths.get("myfeature.json")))
        .withRepository(localRepo)
        .launchFramework();

fw.waitForStop(0);
```

160.4.3 The Feature Launching Process

The following section defines the process through which the Feature Launcher must locate, initialize and populate an OSGi framework when launching a feature. Unless explicitly stated implementations may perform one or more parts of this process in a different order to that described in the specification.

160.4.3.1 Feature Decoration

The first stage of launching is to determine the feature that should be launched by running the configured feature decoration handlers.

First the Feature Launcher must execute any registered FeatureDecorator instances in the order that they were registered. The feature returned by each decorator is used as input to the next.

Once the decoration is complete the Feature Launcher must iterate through the Feature Extensions defined by the feature. For each Feature Extension the launcher must:

1. Identify the Feature Extension Handler for the named extension.
2. If no Feature Extension Handler can be found, and the extension name is one of:
   - LAUNCH_FRAMEWORK
   - FRAMEWORK_LAUNCHING_PROPERTIES
   - BUNDLE_START_LEVELS

   then create an empty Feature Extension Handler which may validate the FeatureExtension.Type of the extension and must return the original feature.

Licensed under the Eclipse Foundation Specification License v1.0. Copyright © Contributors to the Eclipse Foundation.
DRAFT

3. If no Feature Extension Handler has been found or created then check the FeatureExtension.Kind of the extension. If it is MANDATORY then the launch fails with a LaunchException
4. Otherwise call the Feature Extension Handler, and use its result as input when calling any subsequent Feature Extension Handlers.

If any of the decorators throws an AbandonOperationException then the launch operation must immediately fail.

160.4.3.2 Locating a framework implementation

Before a framework instance can be created the Feature Launcher must identify a suitable implementation using the following search order:

1. If any provider specific configuration has been given to the Feature Launcher implementation then this should be used to identify the framework.
2. If the Feature declares an Extension LAUNCH_FRAMEWORK then the Feature Launcher implementation must use the first listed artifact that can be found in any configured Artifact Repositories, as described in *Selecting a framework implementation* on page 99.
3. If no framework implementation is found in the previous steps then the Feature Launcher implementation must search the classpath using the Thread Context Class Loader, or, if the Thread Context Class Loader is not set, the Class Loader which loaded the caller of the Feature Launcher's launch method. The first suitable framework instance located is the instance that will be used.
4. In the event that no suitable OSGi framework can be found by any of the previous steps then the Feature Launcher implementation may provide a default framework implementation to be used.

If no suitable OSGi framework implementation can be found then the Feature Launcher implementation must throw a LaunchException.

160.4.3.3 Creating a Framework instance

Once a suitable framework implementation has been located the Feature Launcher implementation must create and initialize a framework instance. Implementations are free to use implementation specific mechanisms for framework implementations that they recognise. The result of this initialization must be the same as if the Feature Launcher used the org.osgi.framework.launch.FrameworkFactory registered by the framework implementation to create the framework instance.

When creating the framework any framework launch properties defined in the Feature must be used. These are defined as described in *Providing Framework Launch Properties* on page 99 and must include any necessary variable replacement as defined by *Overriding Feature variables* on page 96.

Once instantiated the framework must be initialised appropriately so that it has a valid BundleContext. Once initialised the framework is ready for the Feature Launcher implementation to begin populating the framework.

160.4.3.4 Installing bundles and configurations

The Feature Launcher must iterate through the list of bundles in the feature, installing them in the same order that they are declared in the feature. If bundle start levels have been defined, as described in *Setting the bundle start levels* on page 96, then the Feature Launcher must ensure that the start level is correctly set for each installed bundle. If no start level metadata or extension is defined then all bundles are installed with the framework default start level.

If the installation of a bundle fails because it is determined by the framework to be a duplicate of an existing bundle then the Feature Launcher must treat the installation as a success. The start level of

Licensed under the Eclipse Foundation Specification License - Copyright (c) the Contributors to the Eclipse Foundation.
DRAFT

such a bundle must be set to the lower of its current value and the start level defined for the feature bundle that failed to install.

If a Feature defines one or more Feature Configurations then these cannot be guaranteed to be made available until the ConfigurationAdmin service has been registered. A Feature Launcher implementation may provide an implementation specific way to pre-register configurations, however in general the Feature Launcher should listen for the registration of the ConfigurationAdmin service and immediately create the defined configurations when it becomes available. Configurations must be created in the same order as they are defined in the Feature.

If the CONFIGURATION_TIMEOUT configuration property is set to 0, and one or more Feature Configurations are defined in the Feature being installed, then the implementation must throw a LaunchException unless it is capable of pre-registering those configurations in an implementation specific way.

160.4.3.5 Starting the framework

Once all of the the bundles listed in the feature are installed, and any necessary configuration listener is registered, the implementation must start the OSGi framework. This action will automatically start the installed bundles as defined by the initial start level of the framework, and the start levels of the installed bundles.

The Feature Launcher implementation must delay returning control to the caller until all configurations have been created, subject to the timeout defined by CONFIGURATION_TIMEOUT. The default timeout is 5000 milliseconds, and it determines the maximum length of time that the Feature Launcher implementation should wait to begin creating the configurations. A value of -1 indicates that the Feature Launcher implementation must not wait, and must continue immediately, even if the configurations have not yet been created. If it is not possible to begin before the timeout expires then a LaunchException must be thrown.

Finally, if the minimumStartLevel has been set by the BUNDLE_START_LEVELS extension then the Feature Launcher implementation must check the current start level of the framework. If the current start level is less than the value of minimumStartLevel then the framework's start level must be set to this value.

Once the start process is complete the Framework instance must be returned to the caller.

The following failure modes must all result in a LaunchException being thrown:

- A bundle fails to resolve. If one of the installed bundles fails to resolve this is an error *unless* the Feature is not complete. For Features that are not complete resolution failures must be logged, but not cause a failure.
- A resolved bundle fails to start. If one of the resolved bundles fails to start this is an error *unless* the bundle is a fragment or an extension bundle, which the Feature Launcher should not attempt to start.
- A configuration cannot be created. If a configuration cannot be created then this must result in a start failure

If a launching failure is triggered by an exception, for example a BundleException then this must be set as the cause of the LaunchException that is thrown.

160.4.3.6 Cleanup after failure

If the Feature Launcher implementation fails to launch a feature then any intermediate objects must be properly closed and discarded. For example if an OSGi framework instance has been created then it must be stopped and discarded.

Licensed under the Eclipse Foundation Specification License 1.0 - Copyright © contributors to the Eclipse Foundation
DRAFT

160.5 The Feature Runtime Service

The Feature Runtime Service can be thought of as an equivalent of the Feature Launcher for an existing, running OSGi framework. The Feature Runtime Service therefore does not offer any mechanism for launching a framework, but instead allows one or more features to be installed into the running framework. As an OSGi framework is a dynamic environment the Feature Runtime Service also provides snapshots describing the currently installed Features, allows installed Features to be updated, and allows Features to be removed from the system.

An important difference between the Feature Launcher and Feature Runtime Service is that because the Feature Runtime Service allows multiple Features to be installed it must be able to identify and resolve simple conflicts. For example if two Features include the same bundle at different versions then the resolution may be to install only the higher version, or both versions.

160.5.1 Using the Feature Runtime

The Feature Runtime must be registered as a service in the service registry. Management agents that wish to install, manage or introspect Features in the framework must obtain this service. The Feature Service Runtime service must advertise the FeatureRuntime interface.

160.5.1.1 Thread Safety

Instances of the Feature Runtime are Thread Safe, regardless of whether the service is implemented as a singleton or otherwise. Any FeatureRuntime.OperationBuilder instances created by the Feature Runtime are *not* thread safe and must not be shared between threads.

Despite the Operation Builders not being Thread Safe the underlying Feature Runtime must remain Thread Safe, specifically if two Operation Builders complete at the same time then these calls should be handled sequentially such that there are never partially deployed Features present when installing, updating or removing a Feature.

160.5.1.2 Introspecting the installed Features

An important role for any management agent is being able to introspect the system to discover its current state. The Feature Runtime enables this through the getInstalledFeatures() method, which returns a snapshot of the current state of the system.

The returned list of snapshots contains one InstalledFeature entry for each installed Feature, in the order that they were installed, and may be empty if no Features have been installed. If the framework was started using a Feature Launcher from the same implementation as the Feature Runtime then the Feature Runtime may choose to represent the launched Feature in the snapshot list. If the launched Feature is included in the snapshot list then it must set isInitialLaunch() to true. Launch features cannot be removed or updated by the Feature Runtime, and any attempt to do so must throw a FeatureRuntimeException

Each Installed Feature provides:

- The ID of the Feature from getFeatureId()
- The List of InstalledBundle from getInstalledBundles() listing the bundles installed by the Runtime on behalf of the Feature.
- The List of InstalledConfiguration from getInstalledConfigurations() listing the configurations installed by the feature.

The InstalledBundle snapshots each represent a bundle installed by the Feature Runtime on behalf of the Feature. The Installed Bundle contains the following information:

- getBundleId() - The ID of the bundle that was installed.

Licensed under the Eclipse Foundation Specification License – v1.0. Copyright © Contributing Members of Eclipse Foundation.
DRAFT

- getAliases() - A Collection of one or more IDs that are known to correspond to this bundle. This list will always contain the bundleId and may contain additional IDs if their attempted installation resulted in a collision.
- getBundle() - The actual bundle that was installed into the runtime.
- getStartLevel() - The calculated start level for this bundle. Note that this start level may have been affected by other features.
- getOwningFeatures() - A List of the ids of the features which *own* the installed bundle. Ownership of a bundle is tracked by the Feature Runtime, and it is used to identify when the same bundle forms part of more than one Feature. Bundles that are owned by more than one Feature will not be removed until *all* of the Features that own them are removed.

  In the case where a bundle was not installed by the Feature Runtime, but later became owned by an installed Feature, that bundle will also be owned by the EXTERNAL_FEATURE_ID to indicate that they will not be removed if the other owning Feature is removed.

In addition to bundles Features can contain configurations. The InstalledConfiguration snapshots each represent a configuration created by the Feature Runtime on behalf of the Feature. The Installed Configuration contains the following information:

- getPid() - The configuration pid of this configuration.
- getFactoryPid() - The factory pid of this configuration, or an empty Optional if the configuration is not a factory configuration.
- getProperties() - The merged configuration properties that result from the full set of installed Features contributing to this configuration. Note that there is no dynamic link to Configuration Admin and so any configuration changes made outside the Feature Runtime will not be reflected.
- getOwningFeatures() - A List of the ids of the features which *own* the configuration. Ownership of a configuration is tracked by the Feature Runtime, and it is used to identify when the same configuration, as defined by its pid, forms part of more than one Feature. Configurations that are owned by more than one Feature will not be removed until *all* of the Features that own them are removed.

  In the case where a configuration was not installed by the Feature Runtime, but later became owned by an installed Feature, that configuration will also be owned by the EXTERNAL_FEATURE_ID to indicate that they will not be deleted if the other owning Feature is removed.

160.5.1.3 Installing a feature

Installing a Feature uses one of the install methods present on the Feature Runtime. These methods allow the caller to provide the Feature to be installed and return an FeatureRuntime.InstallOperationBuilder to allow the caller to configure their installation operation. Configuration of operations includes:

- *Setting variable overrides* on page 105.
- *Setting the available Artifact Repositories* on page 105
- *Feature Decoration* on page 97
- Adding *Merging strategies* on page 105

Once the operation is fully configured then the caller uses the install() method to begin the installation. The end result of installing a Feature is that all of the bundles listed in the Feature are installed, all of the Feature Configurations have been created, all bundles have been marked as persistently started, and the framework start level is at least the minimum level required by the Feature.

Start levels for the bundles in the Feature may be controlled as described in *Setting the bundle start levels* on page 96. If any bundles are installed with a start level higher than the current framework start level then they will be marked persistently started but will not start until the framework start level is changed.

Licensed under the Eclipse Foundation Document License – © Copyright and Contributions to the Eclipse Foundation.
DRAFT

In more complex cases, where multiple features are installed with overlapping bundles or configurations then *Merging strategies* on page 105 will be applied to determine which bundles are installed, and what configuration properties will be used when creating or updating a configuration.

If a failure occurs during the installation of a Feature then the Feature Runtime must make every effort to return the system to its pre-existing state. After a failure no new bundles should be installed, any altered configurations returned to their previous states, and the framework start level should be the same as it was prior to the failed installation.

160.5.1.4 Setting the available Artifact Repositories

As with the Feature Launcher, in order to successfully locate the bundles listed in a feature the Feature Runtime must have access to one or more Artifact Repositories capable of locating the bundles. These Artifact Repositories are configured into each Operation Builder by the user.

A configured Feature Runtime will typically include one or more pre-defined Artifact Repositories. These pre-defined repositories are available to view via the `getDefaultRepositories()`. By default all Operation Builders will have access to these repositories when completing. This behaviour can be changed using the `useDefaultRepositories(boolean)` method.

Additional Artifact Repositories can be added to an Operation Builder by calling the `addRepository(String,ArtifactRepository)` method. The supplied name is used to identify the repository. If the supplied name is already used for an existing Artifact Repository then it will be replaced or, if the supplied Artifact Repository is null, removed. A named Artifact Repository added in this way will override a default Artifact Repository with the same name.

160.5.1.5 Setting variable overrides

As described in *Overriding Feature variables* on page 96 a feature may define zero or more overridable properties which can be used to alter the deployment of the feature. These properties may be configured into each Operation Builder by calling the `withVariables(Map)` method. The supplied `Map` contains the keys and values that will override the variables in the Feature.

160.5.1.6 Merging strategies

Merge operations occur when two or more features reference the same, or similar, items to be installed. The purpose of a merge operation is to avoid unnecessary duplication, and to resolve conflicts.

Merging potentially applies whenever a Feature is installed, updated or removed, and may result in different outcomes depending on the strategy used. All runtime merge functions therefore receive a `MergeOperationType` indicating which type of operation is currently running.

160.5.1.6.1 Merging Bundles

Features may define bundles to be installed by including Feature Bundle entries. If two or more Features include Feature Bundles which have IDs with the same group id and artifact id, but which are not the same, then this situation requires a merge to resolve the possible conflict. Determining whether two IDs are the same is accomplished by checking whether they return equal strings from `toString()`.

When a possible conflict is detected the Feature Runtime must call a `RuntimeBundleMerge` to identify the correct actions to take. These actions include:

- Whether to install the candidate Feature Bundle or not
- Whether to re-designate the ownership of any existing Installed Bundles
- Whether to remove any existing Feature Bundles

Although the obvious time for a bundle merge operation to occur is during an `INSTALL` operation, merges may also occur during `UPDATE` and `REMOVE` operations. During an `UPDATE` the existing bundles from the Feature being updated will remain available so that the updated Feature may be

Licensed under the Eclipse Foundation Specification License – v1.0. Copyright © 2012 the Eclipse Foundation
DRAFT

merged into the existing runtime. During a REMOVE a merge will occur to allow Feature ownership to be re-allocated if a shared bundle is being removed.

Merges are resolved by the mergeBundle method which receives:

- The type of the operation, one of INSTALL, UPDATE or REMOVE.
- The Feature being operated on
- The Feature Bundle which requires merging
- A Collection of Installed Bundles representing the currently installed bundles which have an overlapping groupId and artifactId. Note that in the case of an UPDATE or REMOVE operation the Feature being updated or removed will not be present in the collection of owning features for any of the Installed Bundles.
- A List of RuntimeBundleMerge.FeatureBundleDefinition representing the existing Features which form part of the merge operation. Note that in the case of an UPDATE or REMOVE operation the Feature Bundle being updated or removed will not be present in the list. Entries in the list are present in the order that the Features were installed into the runtime.

The result of the merge function is a Stream of RuntimeBundleMerge.BundleMapping. Each Bundle Mapping links a bundle ID to List of feature IDs. The Bundle Mapping's bundle id must only be a bundleId found in the list of Installed Bundles or, in the case of an INSTALL or UPDATE operation, the id of the Feature Bundle being merged. The mapped Feature ids must contain the id of every Feature in the supplied Feature Bundle Definitions, and, in the case of an INSTALL or UPDATE operation, the id of the Feature being merged. If the id of any Installed Bundle is not present in the returned Stream then that bundle will be removed as part of the ongoing operation. If the same bundle id is present more than once the the two mappings will be combined using the union of the mapped Feature ids.

A simple example of a merge strategy which combines configurations by upgrading Features to the highest compatible version could be implemented as follows:

```
public Map<ID,List<ID>> mergeBundle(MergeOperationType operation,
  Feature feature, FeatureBundle toMerge,
  List<InstalledBundle> installedBundles,
  Map<FeatureBundle,Feature> existingFeatureBundles) {

 Map<ID,List<ID>> result;

 if (operation == MergeOperationType.REMOVE) {
  // Just keep everything the same
  result = installedBundles.stream()
    .filter(i -> !i.getOwningFeatures().isEmpty())
    .collect(Collectors.toMap(i -> i.getBundleId(),
      i -> i.getOwningFeatures()));
 } else {
  // Find the Installed bundles we might replace
  Version v = RuntimeMerges.getOSGiVersion(toMerge.getID());

  List<InstalledBundle> sameMajor = new ArrayList<>();
  List<InstalledBundle> differentMajor = new ArrayList<>();

  installedBundles.forEach(i -> {
   if (i.getBundle().getVersion().getMajor() == v.getMajor()) {
    sameMajor.add(i);
   } else {
    differentMajor.add(i);
   }
```

Licensed under the Eclipse Foundation Specification License - v1.0. Copyright © The Eclipse Foundation. All rights to the Eclipse Foundation.
DRAFT

```
  });

  // Bundles with a different major version stay the same
  result = differentMajor.stream()
    .filter(i -> !i.getOwningFeatures().isEmpty())
    .collect(Collectors.toMap(i -> i.getBundleId(),
      i -> i.getOwningFeatures()));

  // Find the biggest existing version and see if it's bigger than v
  Optional<InstalledBundle> max = sameMajor.stream()
    .max((a, b) -> a.getBundle().getVersion()
      .compareTo(b.getBundle().getVersion()))
    .filter(m -> m.getBundle().getVersion().compareTo(v) >= 0);

  // Use the old version if it's bigger, or the new if not
  ID key = max.isPresent() ? max.get().getBundleId() : toMerge.getID();

  Stream<ID> featureIds = sameMajor.stream()
    .flatMap(i -> i.getOwningFeatures().stream());

  result.put(key,
    Stream.concat(Stream.of(feature.getID()), featureIds)
      .collect(Collectors.toList()));
 }
 return result;
}
```

160.5.1.6.2 Merging Configurations

Features may define configurations by including Feature Configuration entries. If two or more Features include properties for the same configuration PID then this situation requires a merge to resolve the conflict.

Merges are resolved by a RuntimeConfigurationMerge which receives:

- The type of the operation, one of INSTALL, UPDATE or REMOVE.
- The Feature being operated on
- The Feature Configuration which requires merging
- The Installed Configuration representing the current state of the configuration. Note that in the case of an UPDATE or REMOVE operation the Feature being updated or removed will not be present in the list of owning features.
- A List of RuntimeConfigurationMerge.FeatureConfigurationDefinition representing the existing Features which form part of the merge operation. Note that in the case of an UPDATE or REMOVE operation the Feature Configuration being updated or removed will not be present in the list. Entries in the list are present in the order that the Features were installed into the runtime.

The result of the merge function is a map of configuration properties that should be used to update the configuration. If the map is null then the configuration should be deleted.

A simple example of a merge strategy which combines configurations by overlaying each in turn and ignoring null configurations could be implemented as follows:

```
public Map<String,Object> mergeConfiguration(MergeOperationType operation,
  Feature feature, FeatureConfiguration toMerge, InstalledConfiguration configuration,
  List<FeatureConfigurationDefinition> existingFeatureConfigurations) {
```

Licensed under the Eclipse Foundation Specification License v1.0. Copyright © Contributors to the Eclipse Foundation.
DRAFT

```
 boolean addedSomething = false;

 Map<String,Object> result = new HashMap<>();

 for (FeatureConfigurationDefinition fcd : existingFeatureConfigurations) {
  FeatureConfiguration fc = fcd.getFeatureConfiguration();
  if(fc.getValues() != null) {
   result.putAll(fc.getValues());
   addedSomething = true;
  }
 }

 if(operation != MergeOperationType.REMOVE && toMerge.getValues() != null) {
  result.putAll(toMerge.getValues());
  addedSomething = true;
 }

 return addedSomething ? result : null;
}
```

160.5.1.7 Removing a Feature

Removing a feature from the Feature Runtime Service uses the remove(ID) method to uninstall and remove a feature from the framework. Removing a feature is a comparatively simple operation, and therefore does not require the configuration of an FeatureRuntime.OperationBuilder.

Once the remove method returns the feature will have been removed from the Feature Runtime, and any links to installed bundles and configurations will have been removed. If this leaves any installed bundles or installed configurations with no owners then these will be uninstalled or deleted from the system as appropriate.

If a failure occurs during the removal of a feature then the Feature Runtime must make every effort to fully remove the feature, for example by continuing to remove installed bundles that no longer have any owners. Exceptions that occur must be logged, and upon completion the Feature Runtime must throw a FeatureRuntimeException which indicates the incomplete removal.

It is not an error to remove a feature which does not exist in the Feature Runtime and this must return without error, and without altering the state of the system. It is an error to attempt to remove any feature that returns true for isInitialLaunch(), and any attempt to do so must result in a FeatureRuntimeException.

160.5.1.8 Updating a Feature

Updating a Feature uses one of the update methods present on the Feature Runtime. These methods allow the caller to indicate which feature should be updated, and provider the new Feature definition to replace it with. The methods return an FeatureRuntime.UpdateOperationBuilder to allow the caller to configure their update operation. Configuration of operations includes:

- *Setting variable overrides* on page 105.
- *Setting the available Artifact Repositories* on page 105
- *Feature Decoration* on page 97
- Adding *Merging strategies* on page 105

Once the operation is fully configured then the caller uses the update() method to begin the update. The end result of updating a Feature is that all of the bundles listed in the new Feature are installed, all of the Feature Configurations in the new Feature have been created, all bundles have been marked as persistently started, and the framework start level is at least the minimum level required by the new Feature. In addition, any bundles and configurations from the old Feature that are

Licensed under the Eclipse Foundation Specification License v1.0. Copyright © Contributors to the Eclipse Foundation.
DRAFT

not present in the new Feature will have been removed, and any configurations present in both the old and new Features will have been updated with new any new content.

At a high level an update operation is therefore superficially similar to performing a remove operation followed by an install operation. The key difference, however, is that any bundles and configurations shared by both features, or identified by a merge strategy, will not be removed, and instead will become owned by the new Feature.

As for installation, start levels for the bundles in the new Feature will be determined as described in *Setting the bundle start levels* on page 96. If any bundles are installed with a start level higher than the current framework start level then they will be marked persistently started but will not start until the framework start level is changed.

Where the feature update includes overlapping bundles or configurations then *Merging strategies* on page 105 will be applied to determine which bundles are installed, and what configuration properties will be used when creating or updating a configuration.

If a failure occurs during the update of a Feature then the Feature Runtime must make every effort to return the system to its pre-existing state. After a failure no new bundles should be installed, any altered configurations returned to their previous states, and the framework start level should be the same as it was prior to the failed installation.

160.5.2 The Feature Runtime Behaviour

The following section provides normative requirements for the behaviour of the Feature Runtime when it is used. This includes the necessary end states after installation, update and removal of Features.

160.5.2.1 The Feature installation process

The Feature Installation process has four main phases:

- The feature decoration phase, where the Feature is decorated and validated
- The bundle installation phase, where Feature bundles are installed
- The configuration creation phase, where Feature Configurations are created
- The Feature Start phase, where Bundles are started.

The feature decoration phase must complete before any other phases can begin. The the bundle installation phase and the configuration creation phase may happen in any order, or even with interleaved steps, however the Feature Start phase must not begin until the bundle installation and configuration creation phases are complete.

160.5.2.1.1 Feature Decoration

The first stage of the operation is to determine the feature that should be used by running the configured feature decoration handlers.

First the Feature Runtime must execute any registered FeatureDecorator instances in the order that they were registered. The feature returned by each decorator is used as input to the next.

Once the decoration is complete the Feature Runtime must iterate through the Feature Extensions defined by the feature. For each Feature Extension the Feature Runtime must:

1. Identify the Feature Extension Handler for the named extension.
2. If no Feature Extension Handler can be found, and the extension name is one of:
   - LAUNCH_FRAMEWORK
   - FRAMEWORK_LAUNCHING_PROPERTIES
   - BUNDLE_START_LEVELS

   then create an empty Feature Extension Handler which may validate the FeatureExtension.Type of the extension and must return the original feature.

Licensed under the Eclipse Foundation Specification License v1.0. Copyright © Contributors to the Eclipse Foundation.
DRAFT

3. If no Feature Extension Handler has been found or created then check the FeatureExtension.Kind of the extension. If it is MANDATORY then the operation fails with a FeatureRuntimeException
4. Otherwise call the Feature Extension Handler, and use its result as input when calling any subsequent Feature Extension Handlers.

If any of the decorators throws an AbandonOperationException then the operation must immediately fail.

160.5.2.1.2 Bundle Installation

When a feature is being installed the Feature Runtime identifies the bundles to be installed. The Feature Runtime also gathers the set of bundles that are already installed, and then computes the overlap between these. Bundles are deemed to overlap if they have the same group id, artifact id, type and classifier but they may differ in version.

If the overlap list contains entries which overlap exactly, that is they have the same version in the runtime and the Feature being installed, then those bundles are removed from the list of bundles to be installed and the existing bundles are marked as *owned* by the Feature being installed. If the marked bundles were not previously owned by any other feature then they also marked as owned by the EXTERNAL_FEATURE_ID to indicate that they will not be removed if the Feature being installed is removed.

Any remaining overlap entries are processed according to the merge strategy for the feature, as described in *Merging Bundles* on page 105. The final list of bundles to install, which excludes any already installed bundles, is then installed in the same order as it was defined by the feature. Each bundle in the feature, including bundles that were already installed, is then marked as owned by the installing feature.

If the installation of a bundle fails because it is determined by the framework to be a duplicate of an existing bundle then the Feature Runtime must treat the installation as a success and add the ID as an alias for the existing Installed Bundle. The start level of such a bundle must be set to the lower of its current value and the start level defined for the feature bundle that failed to install.

Once the installation of bundles is complete the Feature Runtime must uninstall any bundles which were identified for removal as part of any merge processes.

160.5.2.1.3 Configuration Creation

As part of the initial Feature installation the Feature Runtime must also process and create any Feature Configurations that are defined in the Feature. Feature Configurations cannot be guaranteed to be made available until a ConfigurationAdmin service has been registered. A Feature Runtime implementation should therefore listen for the registration of a ConfigurationAdmin service and immediately create or update any pending configurations when it becomes available. Configurations must be created or updated in the same order as they are defined in the Feature.

If the same configuration, as identified by its configuration pid, is defined in one or more existing installed Features then the configuration properties to be used are determined by merging the previous configuration properties with the new properties defined in the Feature, as described in *Merging Configurations* on page 107. If at the point where the FeatureRuntime attempts to create or update a Feature Configuration there are already configuration properties defined in ConfigurationAdmin then these must be ignored and replaced using updateIfDifferent(Dictionary) unless the Configuration is marked as READ_ONLY. If a READ_ONLY configuration does exist then the Feature Runtime must log a warning and skip that configuration.

160.5.2.1.4 Feature Start

Once all of the bundles listed by the feature are installed then the bundles' start levels are assigned as described in *Setting the bundle start levels* on page 96. This includes any pre-existing bundles and the results of any merge operations. If no start level configuration is defined in the fea-

Licensed under the Eclipse Foundation Specification License - v1.0. Copyright © Contributors to the Eclipse Foundation.
DRAFT

ture for a particular bundle then the start level for that bundle is set to the current start level of the framework.

The Feature Runtime must then identify the lowest start level referenced in the Feature, and repeatedly run through the list of bundles, in the order that they are defined in the Feature, looking for bundles which match the identified start level. For each bundle the Feature Runtime must:

- If the bundle was installed in the Bundle Installation phase then set the start level for the bundle.
- If the bundle was already installed then update the start level for the bundle if, and only if, the new start level is lower than the existing start level.
- Mark the bundle as persistently started unless it is a *fragment* bundle.

The Feature Runtime must then identify the next lowest start level referenced in the Feature and repeat this process until all bundles have been persistently started. Once this process is complete then the framework start level must be increased to the minimum start level required by the Feature, or returned to the original framework start level if this is higher and was decreased as part of *Merging Bundles* on page 105.

160.5.2.1.5 Failure scenarios

The following is a non-exhaustive list of possible failure scenarios that must be handled.

- The feature being installed is already known to the Feature Runtime. This must be treated as a failure as the configuration of the InstallOperationBuilder may not be the same as the previous installation. The Feature Runtime must make no changes and immediately throw a FeatureRuntimeException.
- A Feature Bundle cannot be found by any configured ArtifactRepository.
- A BundleException is thrown during *Bundle Installation* on page 110.
- A BundleException is thrown during *Feature Start* on page 110.
- A Feature Configuration cannot be created by the ConfigurationAdmin service.
- An Exception is thrown by any configured ArtifactRepository, RuntimeBundleMerge or RuntimeConfigurationMerge.

In all cases the first exception must be treated as a failure, with the installation process halting immediately. The feature must then be removed from the runtime in a similar manner to calling remove for the feature id. Once the feature removal is complete the failure may be used in creating the FeatureRuntimeException that must be thrown by this method.

160.5.2.2 The Feature removal process

The Feature removal process has four main phases:

- The feature removal phase, where the feature is removed from the Feature Runtime.
- The bundle stop phase, where Installed Bundles without owners are stopped.
- The configuration deletion phase, where Installed Configurations without owners are removed
- The bundle removal phase, stopped bundles are uninstalled

The the feature removal and bundle stop phases may happen in any order, or even with interleaved steps. The same is true for the configuration deletion phase and the bundle removal phase, however these phases must not begin until the bundle stop phase is complete.

160.5.2.2.1 Feature Removal

Feature removal is a simple operation which removes any reference to the Installed Feature from the Feature Runtime. This includes the list of installed features, and the ownership lists of any Installed Bundles or Installed configurations in the Feature Runtime. After removal is complete the ID of the removed feature should not appear anywhere in the Feature Runtime.

Licensed under the Eclipse Foundation Specification License 1.0. Copyright © Contributors to the Eclipse Foundation
DRAFT

Installed Bundles and Installed Configurations which have zero owners after the removal of the feature are now considered eligible for removal. Their removal processes are described in the next phases.

160.5.2.2.2 Bundle Stop

The Feature Runtime must identify the highest start level set by the list of Installed Features, excluding the Feature being removed. If no start level is defined by this list of features then no action is taken, otherwise the framework start level is set to the newly identified start level.

The list of bundles eligible to be stopped, as determined in *Feature Removal* on page 111, is used to peristently stop any remaining bundles. Bundles that are eligible for removal are stopped in the reverse order in which they were started by *Feature Start* on page 110. This is accomplished by stopping the bundles with the highest start level first, using the reverse order of declaration in the feature where the start level is the same. If an eligible bundle is already stopped due to its start level then it must still be persistently stopped.

160.5.2.2.3 Configuration Removal

Once the *Bundle Stop* on page 112 phase has completed the Feature Runtime may begin removing configurations that are eligible. As with bundles, configurations become eligible for removal if they are no longer owned by any feature. Eligible configurations must be removed in the reverse order of creation, that is the reverse order that they were listed in the feature being removed.

160.5.2.2.4 Bundle Removal

Once the *Bundle Stop* on page 112 phase has completed the Feature Runtime may begin uninstalling bundles from the OSGi framework. These bundles must only be eligible bundles identified and stopped as part of the previous phase. Bundles are uninstalled in reverse installation order, that is the reverse of the order in which they are listed in the feature.

If one or more bundles have been uninstalled, and once all eligible bundles have been uninstalled, the Feature Runtime must refresh the framework wiring by calling `FrameworkWiring.refreshBundles`, passing the list of uninstalled bundles. This will cause the framework to completely remove the uninstalled bundles, and any wirings that link to them.

160.5.2.2.5 Failure scenarios

The following is a non-exhaustive list of possible failure scenarios that must be handled.

- The feature being removed is not known to the Feature Runtime. This must not be treated as a failure, and should simply return immediately.
- One or more `BundleExceptions` are thrown during *Bundle Stop* on page 112. These exceptions should be logged when they occur, but then ignored.
- One or more `BundleExceptions` are thrown during *Bundle Removal* on page 112. These exceptions should be logged when they occur, with the Feature Runtime continuing despite the errors. Once the feature removal is complete the failures may be used in creating the `FeatureRuntimeException` that must be thrown by this method.
- One or more Installed Configurations are missing from the `ConfigurationAdmin` service. These missing configurations should be logged with a warning, but not treated as an error.
- One or more Installed Configurations cannot be deleted missing from the `ConfigurationAdmin` service. These exceptions should be logged when they occur, with the Feature Runtime continuing despite the errors. Once the feature removal is complete the failures may be used in creating the `FeatureRuntimeException` that must be thrown by this method.

The Feature Update Process

The Feature Update Process can be viewed as an interleaved remove and installation operation, following the phases present in both.

Licensed under the Eclipse Runtime Specification License – v1.2 - draft © Contributors to the Eclipse Foundation.
DRAFT

- The feature decoration phase, where the new Feature is decorated and validated
- The feature removal phase, where the existing feature is removed from the Feature Runtime.
- The bundle installation phase, where the new Feature bundles are installed
- The bundle stop phase, where Installed Bundles without owners are stopped.
- The configuration creation and update phase, where the new Feature Configurations are created or updated
- The configuration deletion phase, where Installed Configurations without owners are removed
- The Feature Start phase, where Bundles in the new feature are started.
- The bundle removal phase, stopped bundles are uninstalled

160.5.2.3.1 Decorating the new Feature

Decorating the new feature proceeds exactly as if a new feature is being installed, as described in *Feature Decoration* on page 109.

160.5.2.3.2 Removing the existing Feature

Removing the existing feature proceeds exactly as if a new feature is being removed, as described in *Feature Removal* on page 111.

160.5.2.3.3 Installing the new bundles

Installing the bundles from the new feature proceeds as if a new feature is being removed, as described in *Bundle Installation* on page 110, but with two important differences.

The first important difference is that bundles being installed must be prevented from wiring to bundles that are eligible for removal. This may be accomplished through the use of a Resolver Hook. As the resolver may attempt to resolve bundles at any time this restriction must be enforced by the Feature Runtime until after all of the eligible bundles are uninstalled.

The second important difference is that any Installed Bundles that are eligible for removal are *still available* in the runtime. This means that they *must be considered* when determining whether bundles are already installed, or whether they need to be merged. This may lead to one or more Installed Bundles that were eligible for removal becoming *ineligible* for removal as they become owned by the new feature. Any Installed Bundles for which this is the case must be removed from the list of eligible bundles, and immediately become available for wiring by newly installed bundles.

160.5.2.3.4 Stopping the eligible bundles

Stopping the eligible bundles proceeds exactly as described in *Bundle Stop* on page 112. Note that if the existing feature used start levels then this process will likely result in one or more bundles shared between the old and new features being stopped temporarily.

Care must be taken in this phase to persistently stop all eligible bundles. Failing to do so may result in eligible bundles being accidentally restarted in later phases.

160.5.2.3.5 Creating and Updating Configurations

Creating and updating configurations proceeds as described in *Configuration Creation* on page 110, but with one important difference.

Any Installed Configurations that are eligible for removal are *still available* in the runtime. This means that they *must be considered* when determining whether they need to be merged. This may lead to one or more Installed Configurations that were eligible for removal becoming *ineligible* for removal as they become owned by the new feature. Any Installed Configurations for which this is the case must be removed from the list of eligible configurations.

160.5.2.3.6 Removing Configurations

Removing eligible configurations proceeds exactly as described in *Configuration Removal* on page 112.

DRAFT
Licensed under the Eclipse Intellectual Property Protection Agreement 1.0. Copyright © Contributors to the Eclipse Foundation

160.5.2.3.7 Starting the new feature

Starting the new feature proceeds exactly as described in *Feature Start* on page 110. As all bundles eligible for removal were persistently stopped in an earier phase they will remain stopped during this phase, and must not be started again.

160.5.2.3.8 Uninstalling the eligible bundles

Until the Feature Runtime reaches this phase of an update it must fail by attempting to roll back to the previous feature. Once this phase has been reached this failure mode changes, and the Feature Runtime must retain the new Feature, attempting to continue despite failures.

Removing the eligible bundles proceeds exactly as described in *Bundle Removal* on page 112.

160.5.2.3.9 Failure scenarios

The following is a non-exhaustive list of possible failure scenarios that must be handled.

- The feature being updated is not known to the Feature Runtime. This must not make any changes and should immediately throw a FeatureRuntimeException.
- A Feature Bundle cannot be found by any configured ArtifactRepository.
- A BundleException is thrown during *Installing the new bundles* on page 113. This should result in the imediate failure of the operation, rolling back to the pre-update state, with a FeatureRuntimeException thrown to the caller.
- A BundleException is thrown during *Starting the new feature* on page 114. This should result in the imediate failure of the operation, rolling back to the pre-update state, with a FeatureRuntimeException thrown to the caller.
- A Feature Configuration cannot be created by the ConfigurationAdmin service. This should result in the imediate failure of the operation, rolling back to the pre-update state, with a FeatureRuntimeException thrown to the caller.
- An Exception is thrown by any configured ArtifactRepository, RuntimeBundleMerge or RuntimeConfigurationMerge. This should result in the imediate failure of the operation, rolling back to the pre-update state, with a FeatureRuntimeException thrown to the caller.
- One or more BundleExceptions are thrown during *Stopping the eligible bundles* on page 113. These exceptions should be logged when they occur, but then ignored.
- One or more BundleExceptions are thrown during *Uninstalling the eligible bundles* on page 114. These exceptions should be logged when they occur, with the Feature Runtime continuing despite the errors. Once the feature removal is complete the failures may be used in creating the FeatureRuntimeException that must be thrown by this method.
- One or more Installed Configurations are missing from the ConfigurationAdmin service. These missing configurations should be logged with a warning, but not treated as an error.
- One or more Installed Configurations cannot be deleted missing from the ConfigurationAdmin service. These exceptions should be logged when they occur, with the Feature Runtime continuing despite the errors. Once the feature removal is complete the failures may be used in creating the FeatureRuntimeException that must be thrown by this method.

160.6 Capabilities

The Feature Launcher must provide the following capabilities.

160.6.1 osgi.service Capability

The bundle providing the Feature Runtime service must provide capabilities in the osgi.service namespace representing the services it is required to register. This capability must also declare uses constraints for the relevant service packages:

Licensed under the Eclipse Foundation Specification License – v1.0 Copyright © OSGi Contributors to the Eclipse Foundation.
DRAFT

```
Provide-Capability: osgi.service;
 objectClass:List<String>="org.osgi.service.featurelauncher.runtime.FeatureRuntime";
 uses:="org.osgi.service.featurelauncher.runtime",
 osgi.service;
 objectClass:List<String>="org.osgi.service.featurelauncher.repository.ArtifactRepositoryFac
 uses:="org.osgi.service.featurelauncher.repository"
```

This capability must follow the rules defined for the *osgi.service Namespace* on page 65.

160.7 Security

When Java permissions are enabled, the following security procedures apply.

160.7.1 Required Permissions

Bundles that need to make use of the Feature Runtime or Artifact Repository Factory services must be granted permission to get the relevant service, for example ServicePermission[org.osgi.service.featurelauncher.runtime.FeatureRuntime, GET] so that they may retrieve the service and use it.

Only a bundle that provides a Feature Runtime implementation should be granted ServicePermission[org.osgi.service.featurelauncher.runtime.FeatureRuntime, REGISTER] and ServicePermission[org.osgi.service.featurelauncher.repository.ArtifactRepositoryFactory, REGISTER] to register the services defined by this specification.

The Feature Runtime implementation must also be granted ServicePermission[org.osgi.service.cm.ConfigurationAdmin, GET], AdminPermission[*, execute], AdminPermission[*, lifecycle], AdminPermission[*, metadata], AdminPermission[*, resolve], AdminPermission[*, startlevel], AdminPermission[*, context], as these actions are all required to implement the specification.

160.8 org.osgi.service.featurelauncher

Feature Launcher Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.featurelauncher; version="[1.0,2.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.featurelauncher; version="[1.0,1.1)"
```

160.8.1 Summary

- FeatureLauncher - The Feature launcher is the primary entry point for launching an OSGi framework and set of bundles.
- FeatureLauncher.LaunchBuilder - A builder for configuring and triggering the launch of an OSGi framework containing the supplied feature
- FeatureLauncherConstants - Defines standard constants for the Feature Launcher specification.
- LaunchException - A LaunchException is thrown by the FeatureLauncher if it is unable to:
  - Locate or start an OSGi Framework instance

Licensed under the Eclipse Foundation Software User Agreement. Copyright © Contributors to the Eclipse Foundation.
DRAFT

- Locate the installable bytes of any bundle in a Feature
- Install a bundle in the Feature
- Determine a value for a Feature variable that has no default value defined

160.8.2 public interface FeatureLauncher extends ArtifactRepositoryFactory

The Feature launcher is the primary entry point for launching an OSGi framework and set of bundles. As it is a means for launching a framework it is designed to be used from outside OSGi and therefore should be obtained using the ServiceLoader.

Provider Type Consumers of this API must not implement this type

160.8.2.1 public FeatureLauncher.LaunchBuilder launch(Feature feature)

feature the feature to launch

□ Begin launching a framework instance based on the supplied feature

Returns A running framework instance.

Throws `LaunchException`–

160.8.2.2 public FeatureLauncher.LaunchBuilder launch(Reader jsonReader)

jsonReader a Reader for the input Feature JSON

□ Begin launching a framework instance based on the supplied feature JSON

Returns A running framework instance.

Throws `LaunchException`–

160.8.3 public static interface FeatureLauncher.LaunchBuilder

A builder for configuring and triggering the launch of an OSGi framework containing the supplied feature

LaunchBuilder instances are single use. Once they have been used to launch a framework instance they become invalid and all methods will throw IllegalStateException

160.8.3.1 public Framework launchFramework()

□ Launch a framework instance based on the configured builder

Returns A running framework instance.

Throws `LaunchException`–

`IllegalStateException`– if the builder has been launched

160.8.3.2 public FeatureLauncher.LaunchBuilder withConfiguration(Map<String, Object> configuration)

configuration the configuration for this implementation

□ Configure this LaunchBuilder with the supplied properties.

Returns this

Throws `IllegalStateException`– if the builder has been launched

160.8.3.3 public FeatureLauncher.LaunchBuilder withDecorator(FeatureDecorator decorator)

decorator the decorator to add

□ Add a FeatureDecorator to this LaunchBuilder that will be used to decorate the feature being launched. If called multiple times then the supplied decorators will be called in the same order that they were added to this builder.

DRAFT
Licensed under the Eclipse Foundation Specification License v1.0. Copyright © Contributors to the Eclipse Foundation.

Returns this

Throws NullPointerException– if the decorator is null

IllegalStateException– if the builder has been launched

160.8.3.4 public FeatureLauncher.LaunchBuilder withExtensionHandler(String extensionName, FeatureExtensionHandler extensionHandler)

extensionName the name of the extension to handle

extensionHandler the extensionHandler to add

- Add a FeatureExtensionHandler to this LaunchBuilder that will be used to process the named FeatureExtension if it is found in the Feature being launched. If called multiple times for the same extensionName then later calls will replace the extensionHandler to be used.

Returns this

Throws NullPointerException– if the extension name or decorator is null

IllegalStateException– if the builder has been launched

160.8.3.5 public FeatureLauncher.LaunchBuilder withFrameworkProperties(Map<String, Object> frameworkProps)

frameworkProps the launch properties to use when starting the framework

- Configure this LaunchBuilder with the supplied Framework Launch Properties.

Returns this

Throws IllegalStateException– if the builder has been launched

160.8.3.6 public FeatureLauncher.LaunchBuilder withRepository(ArtifactRepository repository)

repository the repository to add

- Add a repository to this LaunchBuilder that will be used to locate installable artifact data.

Returns this

Throws NullPointerException– if the repository is null

IllegalStateException– if the builder has been launched

160.8.3.7 public FeatureLauncher.LaunchBuilder withVariables(Map<String, Object> variables)

variables the variable placeholder overrides for this launch

- Configure this LaunchBuilder with the supplied variables.

Returns this

Throws IllegalStateException– if the builder has been launched

160.8.4 public final class FeatureLauncherConstants

Defines standard constants for the Feature Launcher specification.

160.8.4.1 public static final String BUNDLE_START_LEVEL_METADATA = "bundleStartLevel"

The name of the metadata property used to indicate the start level of the bundle to be installed. The value must be an integer between 0 and Integer.MAX_VALUE.

160.8.4.2 public static final String BUNDLE_START_LEVELS = "bundle-start-levels"

The name for the FeatureExtension of Type.JSON which defines the start level configuration for the bundles in the feature

Licensed under the Eclipse Foundation Specification License v1.0. Copyright © Contributors to the Eclipse Foundation.
DRAFT

160.8.4.3 public static final String CONFIGURATION_TIMEOUT = "configuration.timeout"

The configuration property used to set the timeout for creating configurations from FeatureConfiguration definitions.

The value must be a Long indicating the number of milliseconds that the implementation should wait to be able to create configurations for the Feature. The default is 5000.

A value of 0 means that the configurations must be created before the bundles in the feature are started. In general this will require the ConfigurationAdmin service to be available from outside the feature.

A value of -1 means that the implementation must not wait to create configurations and should return control to the user as soon as the bundles are started, even if the configurations have not yet been created.

160.8.4.4 public static final String FEATURE_LAUNCHER_IMPLEMENTATION = "osgi.featurelauncher"

The name of the implementation capability for the Feature specification.

160.8.4.5 public static final String FEATURE_LAUNCHER_SPECIFICATION_VERSION = "1.0"

The version of the implementation capability for the Feature specification.

160.8.4.6 public static final String FRAMEWORK_LAUNCHING_PROPERTIES = "framework-launching-properties"

The name for the FeatureExtension of Type.TEXT which defines the framework properties that should be used when launching the feature.

160.8.4.7 public static final String LAUNCH_FRAMEWORK = "launch-framework"

The name for the FeatureExtension which defines the framework that should be used to launch the feature. The extension must be of Type.ARTIFACTS and contain one or more ID entries corresponding to OSGi framework implementations. This extension must be processed even if it is Kind.OPTIONAL or Kind.TRANSIENT.

If more than one framework entry is provided then the list will be used as a priority order when determining the framework implementation to use. If none of the frameworks are present then an error is raised and launching will be aborted.

160.8.4.8 public static final String REMOTE_ARTIFACT_REPOSITORY_BEARER_TOKEN = "token"

The configuration property key used to set the bearer token when creating an ArtifactRepository using FeatureLauncher.createRepository(URI, Map)

160.8.4.9 public static final String REMOTE_ARTIFACT_REPOSITORY_NAME = "name"

The configuration property key used to set the repository name when creating an ArtifactRepository using FeatureLauncher.createRepository(URI, Map)

160.8.4.10 public static final String REMOTE_ARTIFACT_REPOSITORY_PASSWORD = "password"

The configuration property key used to set the repository password when creating an ArtifactRepository using FeatureLauncher.createRepository(URI, Map)

160.8.4.11 public static final String REMOTE_ARTIFACT_REPOSITORY_RELEASES_ENABLED = "release"

The configuration property key used to set that release versions are enabled for an ArtifactRepository using FeatureLauncher.createRepository(URI, Map)

160.8.4.12 public static final String REMOTE_ARTIFACT_REPOSITORY_SNAPSHOTS_ENABLED = "snapshot"

The configuration property key used to set that SNAPSHOT release versions are enabled for an ArtifactRepository using FeatureLauncher.createRepository(URI, Map)

Licensed under the Eclipse Foundation Specification License – 1.0. Copyright © the contributors to the Eclipse Foundation
DRAFT

160.8.4.13 public static final String REMOTE_ARTIFACT_REPOSITORY_TRUST_STORE = "truststore"

The configuration property key used to set the trust store to be used when accessing a remote ArtifactRepository using FeatureLauncher.createRepository(URI, Map)

160.8.4.14 public static final String REMOTE_ARTIFACT_REPOSITORY_TRUST_STORE_FORMAT = "truststoreFormat"

The configuration property key used to set the trust store format to be used when accessing a remote ArtifactRepository using FeatureLauncher.createRepository(URI, Map)

160.8.4.15 public static final String REMOTE_ARTIFACT_REPOSITORY_TRUST_STORE_PASSWORD = "truststorePassword"

The configuration property key used to set the trust store password to be used when accessing a remote ArtifactRepository using FeatureLauncher.createRepository(URI, Map)

160.8.4.16 public static final String REMOTE_ARTIFACT_REPOSITORY_USER = "user"

The configuration property key used to set the repository user when creating an ArtifactRepository using FeatureLauncher.createRepository(URI, Map)

160.8.5 public class LaunchException extends RuntimeException

A LaunchException is thrown by the FeatureLauncher if it is unable to:

- Locate or start an OSGi Framework instance
- Locate the installable bytes of any bundle in a Feature
- Install a bundle in the Feature
- Determine a value for a Feature variable that has no default value defined

160.8.5.1 public LaunchException(String message)

message

- Create a LaunchException with the supplied error message

160.8.5.2 public LaunchException(String message, Throwable cause)

message

cause

- Create a LaunchException with the supplied error message and cause

160.9 org.osgi.service.featurelauncher.annotation

Feature Annotations Package Version 1.0.

This package contains annotations that can be used to require the Feature Service implementation.

Bundles should not normally need to import this package as the annotations are only used at build-time.

160.9.1 Summary

- `RequireFeatureLauncherService` - This annotation can be used to require the Feature implementation.

Licensed under the Eclipse Foundation Specification License – v1.0. Copyright © Contributors to the Eclipse Foundation.
DRAFT

160.9.2 @RequireFeatureLauncherService

This annotation can be used to require the Feature implementation. It can be used directly, or as a meta-annotation.

Retention CLASS

Target TYPE, PACKAGE

160.10 org.osgi.service.featurelauncher.decorator

Feature Launcher Decorator Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.featurelauncher.decorator; version="[1.0,2.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.featurelauncher.decorator; version="[1.0,1.1)"
```

160.10.1 Summary

- AbandonOperationException - An AbandonOperationException is thrown by a FeatureDecorator or FeatureExtensionHandler if it needs to prevent the operation from continuing.
- BaseFeatureDecorationBuilder - The BaseFeatureDecorationBuilder is used to allow a user to customize a Feature.
- DecoratorBuilderFactory - The Builder Factory can be used to obtain builders for the various entities.
- FeatureDecorator - A FeatureDecorator is used to pre-process a Feature before it is installed or updated.
- FeatureDecorator.FeatureDecoratorBuilder - A reified builder which adds the ability to replace the extensions for the decorated feature
- FeatureExtensionHandler - A FeatureExtensionHandler is used to check and pre-process a Feature based on its FeatureExtensions before the feature is installed or updated.
- FeatureExtensionHandler.FeatureExtensionHandlerBuilder - A reified builder which does not permit extensions to be modified

160.10.2 public final class AbandonOperationException extends Exception

An AbandonOperationException is thrown by a FeatureDecorator or FeatureExtensionHandler if it needs to prevent the operation from continuing. This may be because of a problem detected in the feature, or because an extension has determined that the feature cannot be used in the current environment.

160.10.2.1 public AbandonOperationException(String message)

message

- Create an AbandonOperationException with the supplied error message

160.10.2.2 public AbandonOperationException(String message, Throwable cause)

message

Licensed under the Eclipse Foundation Specification License - v1.0 Copyright © The Contributors to the Eclipse Foundation.
DRAFT

cause

- Create an AbandonOperationException with the supplied error message and cause

160.10.3 public interface BaseFeatureDecorationBuilder<T extends BaseFeatureDecorationBuilder<T>>

<T> the type of the FeatureDecorator, used to parameterize the builder return values

The BaseFeatureDecorationBuilder is used to allow a user to customize a Feature. It is pre-populated with data from the original Feature, and calling any of the setXXX methods will replace the data in that section.

Provider Type Consumers of this API must not implement this type

160.10.3.1 public Feature build()

- Build the Feature. Can only be called once. After calling this method the current builder instance cannot be used any more. and all methods will throw IllegalStateException.

Returns The Feature.

160.10.3.2 public T extends BaseFeatureDecorationBuilder<T> setBundles(List<FeatureBundle> bundles)

bundles The Bundles to add.

- Replace the bundles in the Feature, discarding the current values.

Returns This builder.

160.10.3.3 public T extends BaseFeatureDecorationBuilder<T> setConfigurations(List<FeatureConfiguration> configs)

configs The Configurations to add.

- Replace the Configurations in the Feature, discarding the current values.

Returns This builder.

160.10.3.4 public T extends BaseFeatureDecorationBuilder<T> setVariable(String key, Object defaultValue)

key The key.

defaultValue The default value.

- Set or replace a single variable in the Feature. If a variable with the specified key already exists it is replaced with this one. Variable values are of type: String, Boolean or BigDecimal for numbers.

Returns This builder.

Throws IllegalArgumentException– if the value is of an invalid type.

160.10.3.5 public T extends BaseFeatureDecorationBuilder<T> setVariables(Map<String, Object> variables)

variables to be added.

- Replace all the variables in the Feature, discarding the current values. Variable values are of type: String, Boolean or BigDecimal for numbers.

Returns This builder.

Throws IllegalArgumentException– if a value is of an invalid type.

160.10.4 public interface DecoratorBuilderFactory

The Builder Factory can be used to obtain builders for the various entities.

This is similar to BuilderFactory but does not permit the creation of FeatureBuilder instances.

Provider Type Consumers of this API must not implement this type

Licensed under the Eclipse Foundation Specification License - v1.0. Copyright © Contributors to the Eclipse Foundation.
DRAFT

160.10.4.1 public FeatureArtifactBuilder newArtifactBuilder(ID id)

id The artifact ID for the artifact object being built.

□ Obtain a new builder for Artifact objects.

Returns The builder.

160.10.4.2 public FeatureBundleBuilder newBundleBuilder(ID id)

id The ID for the bundle object being built. If the ID has no type specified, a default type of @{code jar} is assumed.

□ Obtain a new builder for Bundle objects.

Returns The builder.

160.10.4.3 public FeatureConfigurationBuilder newConfigurationBuilder(String pid)

pid The persistent ID for the Configuration being built.

□ Obtain a new builder for Configuration objects.

Returns The builder.

160.10.4.4 public FeatureConfigurationBuilder newConfigurationBuilder(String factoryPid, String name)

factoryPid The factory persistent ID for the Configuration being built.

name The name of the configuration being built. The PID for the configuration will be the factoryPid + '~' + name

□ Obtain a new builder for Factory Configuration objects.

Returns The builder.

160.10.4.5 public FeatureExtensionBuilder newExtensionBuilder(String name, FeatureExtension.Type type, FeatureExtension.Kind kind)

name The extension name.

type The type of extension: JSON, Text or Artifacts.

kind The kind of extension: Mandatory, Optional or Transient.

□ Obtain a new builder for Feature objects.

Returns The builder.

160.10.5 public interface FeatureDecorator

A FeatureDecorator is used to pre-process a Feature before it is installed or updated. This allows the caller to potentially add or remove extensions, alter feature bundles, or edit configurations before the feature is installed or updated.

Note that a FeatureDecorator is *always called for all features* and may change the feature extensions, as well as bundles, configurations and variables.

160.10.5.1 public Feature decorate(Feature feature, FeatureDecorator.FeatureDecoratorBuilder decoratedFeatureBuilder, DecoratorBuilderFactory factory) throws AbandonOperationException

feature the feature to be installed or updated

decoratedFeature-Builder a builder that can be used to produce a decorated feature with updated values

factory - a factory allowing users to create values for use with `decoratedFeatureBuilder`

□ Provides an opportunity to transform a feature before it is installed or updated

DRAFT
Licensed under the Eclipse Foundation Specification License v1.0. Copyright © Contributors to the Eclipse Foundation.

Returns The Feature to be installed. This must either be the same instance as feature or a new object created by calling decoratedFeatureBuilder.build(). Returning any other object is an error that will cause the install or update operation to fail

Throws AbandonOperationException– if the feature installation or update operation must not continue

160.10.6 public static interface FeatureDecorator.FeatureDecoratorBuilder extends BaseFeatureDecorationBuilder<FeatureDecorator.FeatureDecoratorBuilder>

A reified builder which adds the ability to replace the extensions for the decorated feature

Provider Type Consumers of this API must not implement this type

160.10.6.1 public FeatureDecorator.FeatureDecoratorBuilder setExtensions(List<FeatureExtension> extensions)

extensions The extensions to add.

- Replace the extensions in the Feature, discarding the current values.

Returns This builder.

160.10.7 public interface FeatureExtensionHandler

A FeatureExtensionHandler is used to check and pre-process a Feature based on its FeatureExtensions before the feature is installed or updated. This allows the caller to potentially alter feature bundles, or edit configurations before the feature is installed or updated.

Note that a FeatureExtensionHandler is *only called for features with a matching extension* called and may only change the feature bundles or feature configurations.

160.10.7.1 public Feature handle(Feature feature, FeatureExtension extension, FeatureExtensionHandler.FeatureExtensionHandlerBuilder decoratedFeatureBuilder, DecoratorBuilderFactory factory) throws AbandonOperationException

feature the feature to be installed or updated

extension the feature extension which caused this handler to be called

decoratedFeatureBuilder a builder that can be used to produce a decorated feature with updated values

factory - a factory allowing users to create values for use with decoratedFeatureBuilder

- Provides an opportunity to transform a feature before it is installed or updated

Returns The Feature to be installed. This must either be the same instance as feature or a new object created by calling decoratedFeatureBuilder.build(). Returning any other object is an error that will cause the install or update operation to fail

Throws AbandonOperationException– if the feature installation or update operation must not continue

160.10.8 public static interface FeatureExtensionHandler.FeatureExtensionHandlerBuilder extends BaseFeatureDecorationBuilder<FeatureExtensionHandler.FeatureExtensionHandlerBuilder

A reified builder which does not permit extensions to be modified

Provider Type Consumers of this API must not implement this type

160.11 org.osgi.service.featurelauncher.repository

Licensed under the Eclipse Foundation Specification License 1.0. Copyright © Contributors to the Eclipse Foundation
DRAFT

Feature Launcher Repository Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.featurelauncher.repository; version="[1.0,2.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.featurelauncher.repository; version="[1.0,1.1)"
```

160.11.1 Summary

- `ArtifactRepository` - An ArtifactRepository is used to get hold of the bytes used to install an artifact.
- `ArtifactRepositoryFactory` - A ArtifactRepositoryFactory is used to create implementations of ArtifactRepository for one of the built in repository types:
  - Local File System
  - HTTP repository

160.11.2 public interface ArtifactRepository

An ArtifactRepository is used to get hold of the bytes used to install an artifact. Users of this specification may provide their own implementations for use when installing feature artifacts. Instances must be Thread Safe.

Concurrency Thread-safe

160.11.2.1 public InputStream getArtifact(ID id)

id the id of the artifact

□ Get a stream to the bytes of an artifact

Returns an InputStream containing the bytes of the artifact or null if this repository does not have access to the bytes

160.11.3 public interface ArtifactRepositoryFactory

A ArtifactRepositoryFactory is used to create implementations of ArtifactRepository for one of the built in repository types:

- Local File System
- HTTP repository

Provider Type Consumers of this API must not implement this type

160.11.3.1 public ArtifactRepository createRepository(Path path)

path a path to the root of a Maven Repository Layout containing installable artifacts

□ Create an ArtifactRepository using the local file system

Returns an ArtifactRepository using the local file system

Throws `IllegalArgumentException` – if the path does not exist, or exists and is not a directory

`NullPointerException` – if the path is null

DRAFT
Licensed under the Eclipse Foundation Specification License V1.0. Copyright © The Contributors to the Eclipse Foundation.

160.11.3.2 public ArtifactRepository createRepository(URI uri, Map<String, Object> props)

uri the URI for the repository. The http, https and file schemes must be supported by all implementations.

props the configuration properties for the remote repository. See FeatureLauncherConstants for standard property names

□ Create an ArtifactRepository using a remote Maven repository.

Returns an ArtifactRepository using the local file system

Throws IllegalArgumentException– if the uri scheme is not supported by this implementation

NullPointerException– if the path is null

160.12 org.osgi.service.featurelauncher.runtime

Feature Launcher Runtime Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.featurelauncher.runtime; version="[1.0,2.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.featurelauncher.runtime; version="[1.0,1.1)"
```

160.12.1 Summary

- FeatureRuntime - The Feature runtime service allows features to be installed and removed dynamically at runtime.
- FeatureRuntime.InstallOperationBuilder - The OperationBuilder for a FeatureRuntime.install(Feature) operation.
- FeatureRuntime.OperationBuilder - An OperationBuilder is used to configure the installation or update of a Feature by the FeatureRuntime.
- FeatureRuntime.UpdateOperationBuilder - The OperationBuilder for a FeatureRuntime.install(Feature) operation.
- FeatureRuntimeConstants - Defines standard constants for the Feature Runtime.
- FeatureRuntimeException - A FeatureRuntimeException is thrown by the FeatureRuntime if it is unable to:
  - Locate the installable bytes of any bundle in a Feature
  - Install a bundle in the Feature
  - Determine a value for a Feature variable that has no default value defined
  - Successfully merge a feature with the existing environment
- InstalledBundle - An InstalledBundle represents a configuration that has been installed as a result of one or more feature installations.
- InstalledConfiguration - An InstalledConfiguration represents a configuration that has been installed as a result of one or more feature installations.
- InstalledFeature - An InstalledFeature represents the current state of a feature installed by the FeatureRuntime.
- MergeOperationType - An MergeOperationType represents the type of operation that is in flight

Licensed under the Eclipse Foundation Specification License – v1.0 Copyright © Contributors to the Eclipse Foundation
DRAFT

- RuntimeBundleMerge - Merge operations occur when two or more features reference the same (or similar) items to be installed.
- RuntimeBundleMerge.BundleMapping - A BundleMapping is used to define that a bundle should be (or remain) installed and which Features should own it
- RuntimeBundleMerge.FeatureBundleDefinition - A FeatureBundleDefinition is used to show which FeatureBundle(s) are being merged, and the Feature that they relate to.
- RuntimeConfigurationMerge - Merge operations occur when two or more features reference the same (or similar) items to be installed.
- RuntimeConfigurationMerge.FeatureConfigurationDefinition - A FeatureConfigurationDefinition is used to show which FeatureConfiguration(s) are being merged, and the Feature that they relate to.
- RuntimeMerges - Merge operations occur when two or more features reference the same (or similar) items to be installed.

160.12.2 public interface FeatureRuntime extends ArtifactRepositoryFactory

The Feature runtime service allows features to be installed and removed dynamically at runtime.

Concurrency Thread-safe

Provider Type Consumers of this API must not implement this type

160.12.2.1 public Map<String, ArtifactRepository> getDefaultRepositories()

□ Get the default repositories for the FeatureRuntime service. These are the repositories which are used by default when installing or updating features.

This method can be used to select a subset of the default repositories when using an OperationBuilder, or to query for instances manually.

Returns the default repositories

160.12.2.2 public List<InstalledFeature> getInstalledFeatures()

□ Get the features that have been installed by the FeatureRuntime service

Returns a list of installed features

160.12.2.3 public FeatureRuntime.InstallOperationBuilder install(Feature feature)

feature the feature to launch

□ Install a feature into the runtime

Returns An OperationBuilder that can be used to set up the installation of this feature

Throws LaunchException– if installation fails

160.12.2.4 public FeatureRuntime.InstallOperationBuilder install(Reader jsonReader)

jsonReader a Reader for the input Feature JSON

□ Install a feature into the runtime based on the supplied feature JSON

Returns An installedFeature representing the results of installing this feature

Throws LaunchException– if installation fails

160.12.2.5 public void remove(ID featureId)

featureId the feature id

□ Remove an installed feature

DRAFT
Licensed under the Eclipse Foundation Specification License – 1.0. Copyright © Contributors to the Eclipse Foundation.

160.12.2.6 public FeatureRuntime.UpdateOperationBuilder update(ID featureId, Feature feature)

featureId the id of the feature to update

feature the feature to launch

- Update a feature in the runtime

Returns An installedFeature representing the results of updating this feature

160.12.2.7 public FeatureRuntime.UpdateOperationBuilder update(ID featureId, Reader jsonReader)

featureId the id of the feature to update

jsonReader a Reader for the input Feature JSON

- Update a feature in the runtime based on the supplied feature JSON

Returns An installedFeature representing the results of updating this feature

160.12.3 public static interface FeatureRuntime.InstallOperationBuilder extends FeatureRuntime.OperationBuilder<FeatureRuntime.InstallOperationBuilder>

The OperationBuilder for a FeatureRuntime.install(Feature) operation. Instances are not thread safe and must not be shared.

160.12.3.1 public InstalledFeature install()

- An alias for the complete() method

Returns the installed feature

160.12.4 public static interface FeatureRuntime.OperationBuilder<T extends FeatureRuntime.OperationBuilder<T>>

<T>

An OperationBuilder is used to configure the installation or update of a Feature by the FeatureRuntime. Instances are not thread safe and must not be shared.

Once the complete() method is called the operation will be run by the feature runtime and the operation builder will be invalidated, with all methods throwing IllegalStateException.

160.12.4.1 public T extends FeatureRuntime.OperationBuilder<T> addRepository(String name, ArtifactRepository repository)

name the name to use for this repository

repository the repository

- Add an ArtifactRepository for use by this OperationBuilder instance. If an ArtifactRepository is already set for the given name then it will be replaced. Passing a null ArtifactRepository will remove the repository from this operation.

Returns this

Throws IllegalStateException – if the builder has been completed

160.12.4.2 public InstalledFeature complete() throws FeatureRuntimeException

- Complete the operation by installing or updating the feature

Returns An InstalledFeature representing the result of the operation

Throws FeatureRuntimeException – if an error occurs

DRAFT
Licensed under the Eclipse Foundation Specification License – v1.0 Copyright to Contributions to the Eclipse Foundation.

IllegalStateException– if the builder has been completed already

160.12.4.3 public T extends FeatureRuntime.OperationBuilder<T> useDefaultRepositories(boolean include)

include

- Include the default repositories when completing this operation. This value defaults to true. If any ArtifactRepository added using addRepository(String, ArtifactRepository) has the same name as a default repository then the added repository will override the default repository.

Returns this

Throws IllegalStateException– if the builder has been completed

160.12.4.4 public T extends FeatureRuntime.OperationBuilder<T> withBundleMerge(RuntimeBundleMerge merge)

merge

- Use The supplied RuntimeBundleMerge to resolve any bundle merge operations that are required to complete the operation

Returns this

160.12.4.5 public T extends FeatureRuntime.OperationBuilder<T> withConfigurationMerge(RuntimeConfigurationMerge merge)

merge

- Use The supplied RuntimeConfigurationMerge to resolve any configuration merge operations that are required to complete the operation

Returns this

160.12.4.6 public T extends FeatureRuntime.OperationBuilder<T> withDecorator(FeatureDecorator decorator)

decorator the decorator to add

- Add a FeatureDecorator to this OperationBuilder that will be used to decorate the feature. If called multiple times then the supplied decorators will be called in the same order that they were added to this builder.

Returns this

Throws NullPointerException– if the decorator is null

IllegalStateException– if the builder has been launched

160.12.4.7 public T extends FeatureRuntime.OperationBuilder<T> withExtensionHandler(String extensionName, FeatureExtensionHandler extensionHandler)

extensionName the name of the extension to handle

extensionHandler the extensionHandler to add

- Add a FeatureExtensionHandler to this OperationBuilder that will be used to process the named FeatureExtension if it is found in the Feature. If called multiple times for the same extensionName then later calls will replace the extensionHandler to be used.

Returns this

Throws NullPointerException– if the extension name or decorator is null

IllegalStateException– if the builder has been launched

160.12.4.8 public T extends FeatureRuntime.OperationBuilder<T> withVariables(Map<String, Object> variables)

variables the variable placeholder overrides for this launch

- Configure this OperationBuilder with the supplied variables.

DRAFT
Licensed under the Eclipse Foundation Specification License – v1.0. Copyright © Contributors to the Eclipse Foundation.

Returns this

Throws IllegalStateException– if the builder has been completed

160.12.5 public static interface FeatureRuntime.UpdateOperationBuilder extends FeatureRuntime.OperationBuilder<FeatureRuntime.UpdateOperationBuilder>

The OperationBuilder for a FeatureRuntime.install(Feature) operation. Instances are not thread safe and must not be shared.

160.12.5.1 public InstalledFeature update()

- An alias for the complete() method

Returns the updated feature

160.12.6 public final class FeatureRuntimeConstants

Defines standard constants for the Feature Runtime.

160.12.6.1 public static final String EXTERNAL_FEATURE_ID = "org.osgi.service.featurelauncher:external:1.0.0"

The ID of the virtual *external* feature representing ownership of a bundle or configuration that was deployed by another management agent.

160.12.7 public class FeatureRuntimeException extends RuntimeException

A FeatureRuntimeException is thrown by the FeatureRuntime if it is unable to:

- Locate the installable bytes of any bundle in a Feature
- Install a bundle in the Feature
- Determine a value for a Feature variable that has no default value defined
- Successfully merge a feature with the existing environment

160.12.7.1 public FeatureRuntimeException(String message)

message

- Create a LaunchException with the supplied error message

160.12.7.2 public FeatureRuntimeException(String message, Throwable cause)

message

cause

- Create a LaunchException with the supplied error message and cause

160.12.8 public interface InstalledBundle

An InstalledBundle represents a configuration that has been installed as a result of one or more feature installations.

This type is a snapshot and represents the state of the runtime when it was created. It may become out of date if additional features are installed or removed.

Provider Type Consumers of this API must not implement this type

160.12.8.1 public Collection<ID> getAliases()

- Get any known IDs which correspond to the same bundle

DRAFT
Licensed under the Eclipse Foundation Specification License 1.0. Copyright © OSGi Contributors to the Eclipse Foundation

Returns an immutable collection of aliases for this bundle. Always includes the id returned by getBundleId()

160.12.8.2 public Bundle getBundle()

- The actual bundle installed in the framework

Returns the Bundle installed for this getBundleId()

160.12.8.3 public ID getBundleId()

- Get the ID of the bundle that has been installed

Returns the id of the bundle that was installed

160.12.8.4 public List<ID> getOwningFeatures()

- The features responsible for this bundle being installed, in installation order

Returns A list of Feature IDs

160.12.8.5 public int getStartLevel()

- The start level for this bundle

Returns the start level

160.12.9 public interface InstalledConfiguration

An InstalledConfiguration represents a configuration that has been installed as a result of one or more feature installations.

This type is a snapshot and represents the state of the runtime when it was created. It may become out of date if additional features are installed or removed.

Provider Type Consumers of this API must not implement this type

160.12.9.1 public Optional<String> getFactoryPid()

- Get the factory PID of the configuration

Returns the factory PID of this configuration, or an empty optional if this is not a factory configuration

160.12.9.2 public List<ID> getOwningFeatures()

- The features responsible for creating this configuration, in installation order

Returns A list of Feature IDs

160.12.9.3 public String getPid()

- Get the PID of the configuration

Returns the full PID of this configuration

160.12.9.4 public Map<String, Object> getProperties()

- Get the merged configuration properties for this configuration

Returns The properties associated with this configuration, may be null if the configuration should not be created

160.12.10 public interface InstalledFeature

An InstalledFeature represents the current state of a feature installed by the FeatureRuntime.

This type is a snapshot and represents the state of the runtime when it was created. It may become out of date if additional features are installed or removed.

DRAFT
Licensed under the Eclipse Foundation Specification License v1.0. Copyright © Contributors to the Eclipse Foundation.

Provider Type Consumers of this API must not implement this type

160.12.10.1 public ID getFeatureId()

Returns The ID of the installed feature

160.12.10.2 public List<InstalledBundle> getInstalledBundles()

- Get the bundles installed by this feature

Returns A List of the bundles installed by this feature, in the order they were declared by the feature

160.12.10.3 public List<InstalledConfiguration> getInstalledConfigurations()

- Get the configurations installed by this feature

Returns A List of the configurations installed by this feature, in the order they were declared by the feature

160.12.10.4 public boolean isInitialLaunch()

- Is this a feature installed by FeatureLauncher

Returns `true` If this feature was installed as part of a FeatureLauncher launch operation. `false` if it was installed by the FeatureRuntime

160.12.11 enum MergeOperationType

An MergeOperationType represents the type of operation that is in flight

160.12.11.1 INSTALL

An install operation adds a feature to the runtime

160.12.11.2 UPDATE

An update operation replaces one feature with another

160.12.11.3 REMOVE

A remove operation removes a feature from the runtime

160.12.11.4 public static MergeOperationType valueOf(String name)

160.12.11.5 public static MergeOperationType[] values()

160.12.12 public interface RuntimeBundleMerge

Merge operations occur when two or more features reference the same (or similar) items to be installed.

The purpose of a RuntimeBundleMerge is to resolve possible conflicts between FeatureBundle entries and determine which bundle(s) should be installed as a result.

Merge operations happen in one of three scenarios, indicated by the MergeOperationType:

- INSTALL - a feature is being installed
- UPDATE - a feature is being updated
- REMOVE - a feature is being removed

When any merge operation occurs the merge function will be provided with the Feature being operated upon, the FeatureBundle which needs to be merged, a List of the InstalledBundles representing

Licensed under the Eclipse Foundation Specification License - v1.0. Copyright © Contributors to the Eclipse Foundation.
DRAFT

the currently installed bundles applicable to the merge, and a List of FeatureBundleDefinitions representing the FeatureBundles and installed features participating in the merge. All Installed Bundle and Feature Bundle objects will have the same group id and artifact id.

If an UPDATE or REMOVE operation is underway then the Feature being updated or removed will already have been removed from any Installed Bundles and from the list of Feature Bundle Definitions. For an UPDATE this may result in one or more Installed Bundles having an empty list of owning features, and the list of existing installed Feature Bundle Definitions being empty.

The returned result from the merge function must be a full mapping of installed Bundle IDs to Lists of owning Feature ids. This is returned as a Stream of BundleMappings. The combined BundleMapping.owningFeatures in the stream must contain all of the Feature ids from the list of Feature Bundle Definitions, and in the case of an INSTALL or UPDATE operation also the Feature being operated upon. The entries in the returned stream must only contain BundleMapping.bundleIds from the list of Installed Bundles, and in the case of an INSTALL or UPDATE operation the Feature Bundle being merged

.

It is an error for any value in the returned stream to be null, or to have fields set to null or an empty list. In the case of a REMOVE operation it is an error to include the Feature id being operated upon in the returned Bundle Mappings.

160.12.12.1 public Stream<RuntimeBundleMerge.BundleMapping> mergeBundle(MergeOperationType operation, Feature feature, FeatureBundle toMerge, Collection<InstalledBundle> installedBundles, List<RuntimeBundleMerge.FeatureBundleDefinition> existingFeatureBundles)

operation - the type of the operation triggering the merge.

feature The feature being operated upon

toMerge The FeatureBundle in feature that requires merging

installedBundles A read only list of bundles that have been installed as part of previous installations. This list will always contain at least one entry.

existingFeatureBundles An immutable list of FeatureBundleDefinitions which are part of this merge operation. The entries are in the same order as the Features were installed.

This list may be empty in the case of an UPDATE operation. Note that multiple Feature Bundle Definitions may refer to the same bundle ID, or aliases of a single InstalledBundle.

□ Calculate the bundles that should be installed at the end of a given operation.

Bundle Merge operations occur when two or more features reference a bundle with the same group id and artifact id, and the purpose of this method is to identify which bundles should be/remain installed, and which features they should be owned by.

The returned result from the merge function must be a full mapping of installed Bundle IDs to Lists of owning Features. It is an error for the stream to contain a BundleMapping.bundleId which is not the ID of an entry in in the installedBundle list or, in the case of an INSTALL or UPDATE operation, the ID of the toMerge Feature Bundle.

The combined BundleMapping.owningFeatures in the stream must contain all of the Features from the List of Feature Bundle Definitions, and in the case of an INSTALL or UPDATE operation also the Feature being operated upon. In the case of a REMOVE operation it is an error to include the Feature being operated upon in the returned stream.

It is an error for any entry in the returned stream to be, or contain, null or an empty list.

Returns An unordered Stream of BundleMapping entries linking a bundle id to List of owning Feature ids. Each Bundle Mapping represents a bundle that should be installed as a result of this operation. Note that every Feature id *must* appear in the combined BundleMapping.owningFeatures and that the BundleMapping.bundleId may only contain IDs from toMerge or one of the keys from the installed-

DRAFT
Licensed under the Eclipse Foundation Specification License 1.0. Copyright © Contributors to the Eclipse Foundation

Bundles list. If two BundleMapping entries use the same bundle id, or alias, then this is not an error and these mappings will be combined by the implementation.

160.12.13 public static final class RuntimeBundleMerge.BundleMapping

A BundleMapping is used to define that a bundle should be (or remain) installed and which Features should own it

160.12.13.1 public final ID bundleId

The ID of the bundle to be installed

160.12.13.2 public final List<ID> owningFeatures

The List of features which own the bundle

160.12.13.3 public BundleMapping(ID bundleId, List<ID> owningFeatures)

bundleId

owningFeatures

- Create a new BundleMapping

160.12.14 public static interface RuntimeBundleMerge.FeatureBundleDefinition

A FeatureBundleDefinition is used to show which FeatureBundle(s) are being merged, and the Feature that they relate to.

160.12.14.1 public Feature getFeature()

Returns The Feature containing the FeatureBundle

160.12.14.2 public FeatureBundle getFeatureBundle()

Returns The FeatureBundle being merged

160.12.15 public interface RuntimeConfigurationMerge

Merge operations occur when two or more features reference the same (or similar) items to be installed.

The purpose of a RuntimeConfigurationMerge is to resolve possible conflicts between FeatureConfiguration entries and determine what configuration should be created as a result.

Merge operations happen in one of three scenarios, indicated by the MergeOperationType:

- INSTALL - a feature is being installed
- UPDATE - a feature is being updated
- REMOVE - a feature is being removed

When any merge operation occurs the merge function will be provided with the Feature being operated upon, the FeatureConfiguration which needs to be merged, the InstalledConfiguration representing the current configuration, and a list of FeatureConfigurationDefinitions representing the installed features participating in the merge. All Feature Configurations will have the same PID.

If an UPDATE or REMOVE operation is underway then the Feature being updated or removed will already have been removed from the Installed Configuration and the list of existing Feature Configuration Definitions. For an UPDATE this may result in the InstalledConfiguration.getOwningFeatures() being an empty list, and the map of existing installed Feature Configurations being empty.

The returned result from the merge function is a map of configuration properties that should be used to complete the operation. This may be null if the configuration should be deleted.

Licensed under the Eclipse Foundation Specification License V1.0. Copyright © Contributors to the Eclipse Foundation.
DRAFT

160.12.15.1 **public Map<String, Object> mergeConfiguration(MergeOperationType operation, Feature feature, FeatureConfiguration toMerge, InstalledConfiguration configuration, List<RuntimeConfigurationMerge.FeatureConfigurationDefinition> existingFeatureConfigurations)**

operation - the type of the operation triggering the merge.

feature The feature being operated upon

toMerge The FeatureConfiguration in feature that requires merging

configuration The existing configuration that has been installed as part of previous installations. This will represent a configuration with the same PID as toMerge.

Note that this value will be null if the configuration does not exist to differentiate it from an empty configuration dictionary

existingFeatureConfigurations An immutable list of FeatureConfigurationDefinitions which are part of this merge operation. The entries are in the same order as the Features were installed.

This list may be empty in the case of an UPDATE operation. Note that all Feature Configuration Definitions will refer to the same PID, and this will match the PID of toMerge. An immutable map of existing Feature Configurations which are part of this merge operation. The keys in the map are the Feature Configurations involved in the merge and the values are the Features which contain the Feature Configuration.

- Calculate the configuration that should be used at the end of a given operation.

Configuration merge operations occur when two or more features define the same configuration, where configuration identity is determined by the PID of the configuration. The purpose of this function is to determine what configuration properties should be used after the merge has finished.

Returns A map of configuration properties to use. Returning null indicates that the configuration should be deleted.

160.12.16 public static interface RuntimeConfigurationMerge.FeatureConfigurationDefinition

A FeatureConfigurationDefinition is used to show which FeatureConfiguration(s) are being merged, and the Feature that they relate to.

160.12.16.1 **public Feature getFeature()**

Returns The Feature containing the FeatureConfiguration

160.12.16.2 **public FeatureConfiguration getFeatureConfiguration()**

Returns The FeatureBundle being merged

160.12.17 public final class RuntimeMerges

Merge operations occur when two or more features reference the same (or similar) items to be installed.

The purpose of a RuntimeMerges is to provide common merge strategies in an easy to construct way.

160.12.17.1 **public RuntimeMerges()**

160.12.17.2 **public static Version getOSGiVersion(ID id)**

id

- Attempts to turn the version String from an ID into an OSGi version

Licensed under the Eclipse Foundation Specification License – 1.0. Copyright © contributors to the Eclipse Foundation.
DRAFT

Note that this parsing is more lenient than Version.parseVersion(String). It treats the first three segments separated by . characters as possible integers. If they are integers then they represent the major, minor and micro segments of an OSGi version. If any non-numeric segments are encountered, or the end of the string, then the remaining version segments are 0. Any remaining content from the input version string is used as the qualifier.

Returns An OSGi version which attempts to replicate the version from the ID

160.12.17.3 public static RuntimeBundleMerge preferExistingBundles()

- The preferExistingBundles() merge strategy tries to reduce the number of new installations by applying semantic versioning rules. The new bundle is only installed if it has:
  - A different major version from all installed bundles
  - A higher minor version than all other installed bundles with the same major version

Returns the prefer existing merge strategy

160.12.17.4 public static RuntimeConfigurationMerge replaceExistingProperties()

- The replaceExistingProperties() merge strategy simply replaces any existing configuration values with the new values from the new FeatureConfiguration.

  Removal is more complex and relies on the fact that the existingFeatureConfigurations are in installation order. This means that we can descend the list looking for the previous configuration properties and apply them

Returns the replace existing merge strategy

160.13 References

[1] *The Maven 2 Repository Layout*
https://maven.apache.org/repository/layout.html#maven2-repository-layout

[2] *The Data URI scheme*
https://en.wikipedia.org/wiki/Data_URI_scheme

Licensed under the Eclipse Foundation Specification License 1.0. Copyright © Contributors to the Eclipse Foundation.
DRAFT

DRAFT
Licensed under the Eclipse Foundation Specification License – v1.0. Copyright © Contributors to the Eclipse Foundation.

Licensed under the Eclipse Foundation Specification License – v1.0. Copyright © Contributors to the Eclipse Foundation.
DRAFT

DRAFT
Licensed under the Eclipse Foundation Specification License – v1.0. Copyright © Contributors to the Eclipse Foundation.

End Of Document