Mt Wilson

Launcher

# Background

The launcher is responsible for starting the Mt Wilson server. This involves orchestrating various startup tasks such as detecting and loading the configuration, loading available extensions, starting the Jetty web server, and starting other background tasks.

The launcher is configurable to support other applications.

# Architecture

The mtwilson-launcher project contains a Main class for the Java Runtime to execute as the entry point to Mt Wilson. The mtwilson-launcher Main class must load the application properties, load the available extensions, and choose a command to execute based on the command-line arguments.

The default application properties are for Mt Wilson. Other applications can place an “application.properties” file in the classpath under the com.intel.mtwilson package in order to override the defaults.

After the launcher loads the application.properties, or assigns the defaults, it sets them as system properties so they are available for other components to reference. This allows other components and specifically the configuration component to read the properties without a dependency on the mtwilson-launcher project. The launcher respects existing properties so if any system properties are already defined, the launcher does not replace them.

The mtwilson-launcher-api project provides annotations and interfaces that allow any component to hook into the launch process. For example, a component could annotate a Runnable implementation as @Background task and the launcher will run it automatically in a background thread.

# Interfaces

The launcher communicates settings to the rest of the application via the system properties. The launcher respects existing properties and will not override them. This allows the developer customizing an application to override certain properties from the command line.

Environment variables are not used as part of this interface because that would create a situation where they could be inconsistent with system properties.

## Application Properties

An application can define its own settings in a classpath resource named /com/intel/mtwilson/application.properties. For example, Trust Director defines its configuration file name as “director.properties”, while Key Broker defines it as “kms.conf”.

The application.properties file can be provided as a classpath resource by any jar on the classpath, therefore it must only be defined by a jar belonging to the application in order to override the defaults provided by the launcher. Typically it would be included as a classpath resource in the main jar of the application, but in applications that don’t have a main jar, they can simply package a jar containing just this resource and include it in the classpath.

The following table describes the available application properties.

Table Available properties in application.properties

|  |  |  |
| --- | --- | --- |
| Property | Default Value | Notes |
| mtwilson.application.id | mtwilson | Controls prefix for other properties, for example mtwilson.home would become kms.home if id is set to “kms” |
| mtwilson.application.name | Mt Wilson | Display name for the application |
| mtwilson.configuration.file | mtwilson.properties | Name of the configuration file (relative to configuration folder) or absolute path |
| mtwilson.environment.prefix | mtwilson | Prefix for environment variables (MTWILSON\_) and system properties (mtwilson.) |

# Acceptance Tests

First, the original installer still needs to work: installers/mtwilson-server with glassfish.

Second, the new installer should work with jetty.

# Development Roadmap

These are the next steps:

* Project in “packages” folder which has all Mt Wilson java dependencies - this will be copied to the /opt/mtwilson/java folder: mtwilson-server-zip/target/feature/java
* “update package” Script on build server to “update” /opt/mtwilson/java from updated jars in the packages folder (easy rsync)
* “update all” and “update grep <regex>” Script on build server to “update” /opt/mtwilson/java from updated jars throughout the source directory (already have this )
* kms-configuration -> mtwilson-configuration & mtwilson-setup
* MyConfiguration: need to find uses of every public method and replace with v3 equiv. or rewrite internal to use v3 equiv. and mark deprecated
  + MyConfiguration constructor: My.configuration(), mtwilson-test:InitMyConfig, mtwilson-configuration:FilesystemTest
  + MyConfiguration(Properties) constructor: mtwilson-my:MyPersistenceManager
  + getSource: mtwilson-console:CheckConfig and mtwilson-test:CertificateDownloadTest,RpcTest,VmATtestationTest can be replaced with v3 LayeredConfiguration getSource
  + update: mtwilson-my:MyConfigurationTest, mtwilson-setup:CreateTlsCertificate (to save keystore password - new setup v3 has facility for this with configuration provider)
* Remove uses of ConfigurationUtil… mostly test classes, only 3 classes in production, and also restore the all caps environment config…. actually if I just move it to configuration util it will probably work ok.
* Mt wilson command / script to start the server via jetty… so I don’t have to run installer again after every build. just update java and restart. “mtwilson start” will detect jetty if both tomcat and glassfish are missing…
* command to scan extensions should be available via java service api (ok so it’s a setup task - so setup command is also available via java service api). then use it to scan extensions after an update (integrate into update/restart script)
* Remove “SetupManager” command from mtwilson-setup (replaced by Setup command) - currently only referenced by TrustAGent’s Setup which extends it just to change trustagent.properties which is now available from system.getproperty(configuration.file)
* Refactor some of the extensions utilities between scanner/registrars/filters/selectors so that I can have: selectors for nominating classes, filters for rejecting them, and registrars for actually registering ????? something is still wrong with that, but the issue is that an implementation AnnotationRegistrar defines a useful rule that it won’t register java.\* or javax.\* classes, which is something that should go in a filter that can be applied “globally” to any registration. then the user can extend it by adding more package restrictions with includes “only look in com.intel.\*” and/or excludes “ignore anything in org.apache.\*” Also I need another Scanner that instead of nominating every class file for registration, it can scan the bytecode to check for uses of Extensions.find, Extensions.findAll etc. essentially extension points and just grab the specific classes… this will work as long as all extension points are static and it can skip dynamic checkpoints (where a class variable is passed in) or maybe future implementation could attempt to follow dynamic checkpoints to see if they can be statically resolved and if not then skip them and maybe log somewhere the static and dynamic extension points so the admin can see them all.