Intel Cloud Integrity Technology 3.0

**HTML5**

# Background

The mtwilson-core-html5 project has two purposes: first, to provide a common shared stylesheet as the foundation of all Mt Wilson applications with the Intel logo and color scheme; second, to provide a mechanism for Mt Wilson applications to dynamically extend the UI with features.

# Architecture

## Splash Screen

The root path to the application retrieves a “splash screen” that displays the Intel logo and then redirects the browser to the main page of the application.

The splash screen’s style should be consistent with the application style and it contains some elements of the application main page for continuity.

The splash screen is provided by mtwilson-core-html5. It is displayed while the rest of the application continues to load asynchronously.

## Main Application

The mtwilson-core-html5 project contains the static HTML5 resources for the main application. Other features are loaded and inserted to the UI by this code.

The default application title is “Cloud Integrity Technology”. The application title can be changed by setting the application title property to the desired text in the localized translation file, which is part of the internationalization feature.

## Extension Points

The user interface foundation provided by mtwilson-core-html5 has extension points to which applications can add features.

An extension point definition consists of two parts: first, the extension point identifier, which looks like a path; second, the format required for any extensions implemented for that extension point, which can be a file format (js, json, xml, png, html, css, etc.) and the internal format (for a json example, the file must contain a json object and must have a top-level attribute named xyz, etc.)

The main “index.html” has an extension point “/mtwilson-core-html5/init/ready.js”. Any feature that implements this can run its “on ready” javascript similar to having its own $(document).ready(…) event which is triggered after the user logs in.

For example, the main menu has an extension point for new menu items and the footer has an extension point for more links. There is also a “run at load” extension point to allow any feature an opportunity to affect the UI even though there may not be an extension point defined specifically for what it needs to do.

## Discovery

The mtwilson-core-feature project has an API (/features) that the UI can query to discover the list of installed features that extend the HTML5 user interface (/features?extends=html5).

The mtwilson-core-html5 project has an API (/html5/directory) that the UI can query to discover plugins and hooks for a given UI extension point in a specific feature (/html5/directory?feature=mtwilson-core-version&path=/menubar/buttons.json) or in all features (/html5/directory?path=/menubar/buttons.json). The directory listing includes a link to download each file (/html5/features/mtwilson-core-version/menubar/buttons.json).

HTML5 content provided by Mt Wilson features is stored under /opt/mtwilson/features/{featureId}/html5. The download API path starts with /html5/features/{featureId} which is translated to the local disk path. The rest of the URI path is then normalized and appended, ensuring that only HTML5 content within the specified feature can be accessed.

Note that the extension point itself is implicitly provided in the path. For example, looking for a “/menubar/buttons.json” in all features (/html5/directory?path=/menubar/buttons.json) is interpreted to mean that “/menubar/buttons.json” is the name of the extension point, such that the feature making that API call is the menubar and it’s looking for other features that are providing buttons to display in the menubar. There doesn’t need to be a “menubar” Maven project to correspond to it. A single feature may define many extension points. To avoid naming conflicts, it will be helpful for features (especially features not part of Mt Wilson core components) to define their extension points within a namespace, for example “/mtwilson-core-html5/menubar/buttons.js” or “/com.intel.mtwilson.core.html5/menubar/buttons.js” instead of just “/menubar/buttons.js”.

A feature.xml file provides a description of the feature, author information, copyright, and licenses. The feature.xml includes a section called “requires” in which other features are listed that are required to be installed in order for this feature to function. It also includes a section called “conflicts” in which other features are listed that must not be installed in order for this feature to function - typically the conflicting relationship is because the two plugins are alternatives where only one can be active, or is because there is a dependency management issue and the conflict is restricted to a specific version of the software and may be resolved in later versions. The feature.xml includes a section called “extends” in which other features are listed that have extension points implemented by this feature. An “extends” relationship does not imply “requires” - extensions are provided and if the other feature is not installed they will simply not be activated.

The “run at load” extension point (/mtwilson-core-html5/init/ready.js) runs arbitrary javascript provided by matching extensions.

The mtwilson-core-html5 project will detect extension points in each feature.

## Packaging

Projects that have UI content must generate a zip file with the content to be copied to /opt/mtwilson/features/{featureId}/html5. That full path will be generated automatically and the contents of the zip file will be extracted inside it.

See the development section for instructions on creating a feature zip file.

## Linking

The main application will provide an “endpoint” value which could be a relative path or an absolute path to the application itself. All paths specified by features should be relative to this “endpoint” value. When forming links to write into HTML, the format is “endpoint/relative”. The “endpoint” value should not have a trailing slash because all relative paths will add the slash.

Tab identifiers are URI fragments so they should conform to <https://tools.ietf.org/html/rfc3986#section-3.5>

# Formats

## Feature Package

A feature package is a zip file with the following structure:

feature.xml

java/

html5/

The “java” folder should include only the .jar files that implement the feature; dependencies should be declared in the META-INF section of the .jar files but not included directly.

The “html5” folder should include any html, javascript, stylesheets, images, or other static content.

## Feature XML

The feature.xml file describes the feature. The format is defined by “mtwilson-feature-xml”. A short example is included here.

<?xml version="1.0" encoding="UTF-8"?>

<feature xmlns="urn:mtwilson:feature:1.0">

<id>mtwilson-feature-xyz</id>

<version>1.0</version>

<name>Mt Wilson Feature XYZ</name>

<provider><name>Intel Corporation</name></provider>

<description>Illustrates content of feature.xml</description>

<license><copyright>2015 Intel Corporation</copyright><url>http://intel.com</url></license>

</feature>

## Feature Identifier

A feature identifier can be a simple string, or a UUID, or a Java package name.

## Extension Point Identifier

An extension point is identified by a path, starting with a forward-slash “/” and ending in a filename. There can be zero or more directories, each followed by a forward-slash, between the leading forward-slash and the filename. The path corresponds to a URL fragment.

By convention, every extension point includes at least one directory and the first directory is the name of the feature which has defined the extension point. For example, if feature “mtwilson-xyz” defines an extension point for its menubar, the full extension point identifier might be “/mtwilson-xyz/menubar/buttons.json”. Other features implementing extensions to the “mtwilson-xyz” menubar, would have a source file “src/main/html5/mtwilson-xyz/menubar/buttons.json” which would be in whatever format is required by the extension point.

The feature identifier can look like a standard artifact name “mtwilson-xyz” or it can be a fully qualified name in reverse, like Java package names “com.intel.mtwilson.xyz”.

# Development

## How to create a project to build a feature with HTML5 UI elements

### Maven project layout (source)

pom.xml

feature.xml

src/main/html5

src/main/java

And use this parent pom:

<parent>

<groupId>com.intel.mtwilson.maven</groupId>

<artifactId>mtwilson-core-feature-zip</artifactId>

<version>3.0-SNAPSHOT</version>

<relativePath/>

</parent>

The project will generate a .jar with the Java code, a .zip with the HTML5 files, and another feature.zip with both Java and HTML5 content.

### Feature zip file layout (build)

This is the outline of the feature zip file:

feature.xml

html5/

java/

All the content from src/main/html5 in the Maven project will be found under the html5 folder in the zip file.

### Application file layout (deploy)

The feature.xml and the HTML5 content will be found under the feature identifier under the “features” folder:

/opt/mtwilson/bin/

/opt/mtwilson/configuration/

/opt/mtwilson/env/

/opt/mtwilson/features/xyz/feature.xml

/opt/mtwilson/features/xyz/html5/

/opt/mtwilson/java/

/opt/mtwilson/logs/

/opt/mtwilson/repository/

## How to create an application with features

Create an application project and use this parent pom:

<parent>

<groupId>com.intel.mtwilson.maven</groupId>

<artifactId>mtwilson-core-application-zip</artifactId>

<version>3.0-SNAPSHOT</version>

<relativePath/>

</parent>

Then, in the dependencies section, reference feature projects like this:

<dependency>

<groupId>com.intel.mtwilson.core</groupId>

<artifactId>mtwilson-core-version</artifactId>

<version>0.1-SNAPSHOT</version>

<classifier>feature</classifier>

<type>zip</type>

</dependency>

A feature project is one that inherits from mtwilson-core-feature-zip, or uses any other mechanism to create an artifact with classifier “feature” and type “zip” which contains the “java”, “html5”, etc. subfolders described elsewhere in this document.

## How to define an extension point

See the “extension point identifier” section under “formats” for guidance on naming the extension point.

Prefer declarative extensions for well-established behavior such as adding a link, button, or tab to an existing UI feature.

Prefer scripting extensions for unexpected behavior such as hooks before/after events.

Combinations are possible. For example “/feature-xyz/menubar/buttons.json” and “/feature-xyz/content/tabs.json” and “/feature-xyz/content/on\_select\_tab.js” can all be defined to work together.

The global variable “discovery” is defined by the main page (index.html). Features can use the “discovery” mechanism at their extension points to find available extensions.

The “discovery.eachJSON(path, callback)” method invokes the callback for each file that has a matching path across all available features.

## How to enable forward-integration with unknown new features

Define extension points and use discovery to find all features that are designed to integrate with your extension points. Implement hooks by enabling all available features, and implement plugins by selecting a specific feature based on user input.

## How to integrate with existing features

When you need to integrate with existing features, implement the extensions that correspond to their defined extension points. This is a passive method where those features will discover your implementation and use it. An example of the passive method is providing navigation buttons to the navbar feature.

When you need to drive the use of an existing feature, use detection or discovery to determine if it’s available and possibly load it before using it. This is an active method. An example of the active method is querying server configuration for a setting specific to your feature.

Whenever possible, avoid looking for a specific feature by name - instead, discover what features implement a specific interface (extension point).

## How to display available features in consistent order

The discovery mechanism can result in a random order of features displayed each time the user logs in, for example in a navigation menu.

To overcome this, an extension point can define a setting to determine the order of features and use the settings API to get the value of this configured setting. Then, whatever features are available that are mentioned in that list, could be shown in the configured order, and the configuration can also be used to determine whether to allow new features to be shown after the ordered list or not to show new features at all (using the order to also restrict which features are visible).

In the future a more specific “UI Builder” feature could be used to help the users sort features and save/load the order using a separate storage for that information so it doesn’t clutter the application settings.

# Operation

## How to add a feature to an installed server

In the future a package manager would automate this:

* Copy .jar files to /opt/mtwilson/java
* Copy html5 files to /opt/mtwilson/features/{featureId}/html5

Also in the future the .jar file might stay under /opt/mtwilson/features/{featureId}/java and referenced directly from there in the application classpath, so the /opt/mtwilson/java folder wouldn’t be necessary.

# Security

The server features supporting the extensible UI are the installed feature search and the HTML5 feature directory search. These APIs are protected and require a logged-in user for access.

Therefore, the splash page and login page do not support feature discovery.

It is still possible to make the login page somewhat dynamic by loading a list of available plugins from a JSON file in the same folder (that doesn’t require permissions to access), for example to support OpenID or other login that has a different UI entry point.

# Opens

## Using fully qualified feature identifiers

Currently the features identifiers are set to the feature’s Maven artifact identifiers, for example “mtwilson-core-html5”, “mtwilson-configuration-settings-ws-v2”, “pte-logs”, “kms-keys”, etc.

However, the Maven artifact identifiers are within a namespace created by Maven group identifiers, whereas feature identifiers are in a global namespace. To avoid naming conflicts, the feature identifiers should be set to include fully qualified names similar to Java package names. A similar practice is employed in several open source projects that have a plugin capability, for example Eclipse feature identifiers.

With this change, feature identifiers would be within a namespace created by company ownership of a domain name (in reverse domain name notation like “com.intel”).

A regular expression for the reverse domain name notation is provided in Wikipedia:

^[A-Za-z]{2,6}((?!-)[A-Za-z0-9-]{1,63}(?<!-)\\.)+$

<https://en.wikipedia.org/wiki/Reverse_domain_name_notation>

If this change is implemented, it will impact multiple projects that already use the prototype framework, but all touch points can be easily identified by a simple search for each feature id, followed by cascading replacements into the content wherever paths or other names are constructed using the identifier.

Table 1 Design alternatives for feature identifiers

|  |  |  |
| --- | --- | --- |
| **Alternative** | **+** | **-** |
| Use Maven artifact identifier | Easily correlate feature identifier to Maven project | Maven artifact identifiers are within a namespace created by Maven group identifiers, whereas feature identifiers are in global namespace  Maven artifact identifiers use regular expression [A-Za-z0-9\_\\-.]+ which allows them to start with special character and have multiple dots in sequence, which can result in awkward valid names |
| Use fully qualified feature identifiers with reverse domain name notation | Avoid collisions by using reverse domain name as the namespace  Regular expression available and must start with letter and cannot contain a sequence of more than one dot |  |

## Definition of hooks and plugins

Currently each extension is implemented as a separate set of files for discovery and additional resources. This has an impact on network performance.

Considering addition of centralized extensions.js in each feature to describe implemented extensions with links to additional resources, so that client only makes one discovery request for extensions.js at login and then loads additional resources as necessary later.

If the addition is adopted, existing implementations will continue to work but as they are migrated to extensions.js the additional requests will not be necessary as client will already have the answers cached and can avoid the additional request; eventually the original extension files can be removed.

The centralized file would be located in a subdirectory like /opt/mtwilson/features/{featureId}/html5/extensions.js and the content would look like this:

{

"/menubar/buttons.js": [

{

"id": "db355a56-e83c-4a43-9e9e-20f704d56cb5",

"feature": "mtwilson-core-html5",

"href": "/html5/features/mtwilson-core-html5/mtwilson-core-html5/navbar/profile.html"

},

{

"id": "225344c5-7959-4cb4-aa30-cbeb2bf9cf3a",

"feature": "mtwilson-core-html5",

"href": "/html5/features/mtwilson-core-html5/mtwilson-core-html5/navbar/logout.html"

},

{

"id": "03e26d02-7e98-479d-8efd-37a989b77f5c",

"feature": "mtwilson-configuration-settings-ws-v2",

"href": "/html5/features/mtwilson-configuration-settings-ws-v2/mtwilson-core-html5/navbar/settings.html"

}

]

}

The UUIDs can be generated by the client if they are missing from the server response, so the client can use them when it needs to reference something by id on a page.

Table 2 Design alternatives for definition of hooks and plugins

|  |  |  |
| --- | --- | --- |
| **Alternative** | **+** | **-** |
| Decentralized definitions, for example /navbar/hooks.js and /help/main.js discovered separately | Easy to copy/paste skeleton implementation  Easy to see what is implemented by looking at directory structure | Performance impact for discovery of multiple files per feature; one discovery request per extension point (assumes caching on client) + separate download request for each implementation |
| Centralized definitions, for example /extensions.js discovered once and includes keys for “/navbar/hooks.js” and “/help/main.js” | Easy to see and manage all implementations in one file (with json pretty print), and search/replace for changes  Better network performance because only one discovery request (assumes caching on client) + separate download request for each feature as its needed |  |
| Centralized definitions and consolidated content (actual content of “/navbar/hooks.js” and “/help/main.js” included in “included” section) | Best network performance because only one discovery request (assumes caching on client) and no follow-up requests, due to all extensions.js content being attached to discovery response | Client forced to download all extensions data up front; this may be a large amount of data and cause noticeable delay during login even for features that won’t be used |