thingworx®

ThingWorx Extension Development Guide

Version 4.5

June 2018

UNITED STATES GOVERNMENT RIGHTS

PTC software products and software documentation are “commercial items” as that term is defined at 48 C.F.R. 2.101. Pursuant to Federal Acquisition Regulation (FAR) 12.212 (a)-(b) (Computer Software) (MAY 2014) for civilian agencies or the Defense Federal Acquisition Regulation Supplement (DFARS) at 227.7202-1 (a) (Policy) and 227.7202-3 (a) (Rights in commercial computer software or commercial computer software documentation) (FEB 2014) for the Department of Defense, PTC software products and software documentation are provided to the U.S. Government under the PTC commercial license agreement. Use, duplication or disclosure by the U.S. Government is subject solely to the terms and conditions set forth in the applicable PTC software license agreement.

PTC Inc., 140 Kendrick Street, Needham, MA 02494 USA

Document Revision History

Revision Date	Version	Description of Change
December 2, 2015	3.0	Updated version based on the Eclipse Plugin for ThingWorx Extension Development
February 15, 2016	3.1	Added the API Compatibility section
April 18, 2016	4.0	New content including the Widget Development Guide (previously a separate document named *Creating Widgets for ThingWorx*)
August 4, 2016	4.1	Updates based on ThingWorx Core 7.2
December 15, 2016	4.2	Added Entity Characteristic Support section
March 29, 2017	4.3	Added additional best practices.
October 13, 2017	4.4	Updated the *Defining Thing Templates* section to note that marking a Thing Template as an editable extension object is not supported.
June 8, 2018	4.5	Updated Configuration Table information, Localization Table information; added best practices

Contents

1

About this Guide

This guide explains why and how you would develop an extension for ThingWorx. It describes best practices for creating ThingWorx extensions.

Prerequisites

To develop ThingWorx extensions, you must have the following:

Training

- An understanding of the ThingWorx Composer and its modeling capabilities
- An understanding of the ThingWorx Mashup Builder

ThingWorx Platform and Development Tools

- ThingWorx 6.6.0 or newer Platform instance
- Access to the Apache Tomcat instance that is running the ThingWorx instance
- Java SE Development Kit (JDK) 8
- The ThingWorx Extension SDK for your version of ThingWorx

To use the Eclipse Plugin for ThingWorx Extension Development, you must have the following:

Eclipse

- Eclipse IDE for Java EE Developers, Mars 4.5 release or newer
- The Eclipse Plugin for ThingWorx Extension Development

2

Introduction to Extensions

What is an Extension?

An extension is a collection of entities, resources, and widgets that is used to extend the functionality of the ThingWorx Platform. This collection is packaged into a .zip file, which can be imported to any ThingWorx Platform and is used to expose new functionality (such as email servers).

Entities are created using Composer. You can create widgets, resources, and Java code using external tools for writing HTML, CSS, and JavaScript, such as Eclipse.

You can use extensions as building blocks for delivering new services or applications for the ThingWorx Platform. You can deliver these building block extensions individually or you can zip them together for easy deployment.

Extensions can be published on the ThingWorx Marketplace, which is accessible by the customer community of PTC/ThingWorx.

Why Build an Extension?

There are several reasons why you might want to build an extension, including the following:

- Your solution includes multiple entities that depend on one another's existence in order to function.
- Your solution depends on a Java library that is not accessible within ThingWorx Platform.
- You would like to hide your source code from those who use the extension.
- You would like to use a custom widget that does not exist on ThingWorx Platform.
- You want a global service that is not associated with an entity (resource).
- Your organization needs to use a custom directory service or user authorization scheme.

3

Creating Extensions

You can create ThingWorx extensions using common editors and development tools, as long as the artifacts are created following the expected convention and are packaged in the expected structure. This section explains how these artifacts must be created so they work correctly after being imported into ThingWorx.

Tip

Because it can be challenging to correctly build the various pieces of an extension, the Eclipse Plugin for ThingWorx Extension Development (Eclipse Plugin) has been created to help developers build extensions quickly and concentrate on developing their features rather than worrying about getting the extension structured correctly. The productivity improvements provided by the Eclipse Plugin make it a compelling tool to use when building ThingWorx extensions. For more information, see Use the Eclipse Plugin.

Extension Zip File Structure

The high-level process for creating an extension includes a combination of the following steps:

1. Specify the extension information and its artifacts in a metadata.xml file.
2. Model various ThingWorx entities in Composer, and export these entities into their corresponding XML files for inclusion in the extension.
3. Create a JAR file containing Java-based entities and other classes, and include these and any required third-party JAR files in the extension.
4. Create custom widgets to be included in mashups.

These artifacts must be packaged into the extension zip file in the following folder structure:

Zip File Element	Description
`/metadata.xml`	The `metadata.xml` file contains information about the extension and details of the various artifacts within the extension.
`/Entities/`	The `Entities` folder contains zero or more entity XML files organized in subfolders by entity type.
`/lib/`	The `lib` folder contains the JAR file, which includes the custom Java classes written for the extension and third-party JAR files required by the extension.
`/ui/`	The `ui` folder contains the files needed to define custom widgets that are used when building and running mashups.

You can name the zip file anything, but it must contain a valid `metadata.xml` file in the root folder, and its artifacts must be placed in the correct location within the zip file.

Metadata.xml File

The `metadata.xml` file contains information about the extension and details for the various artifacts within the extension. The information in this file is used in the import process in ThingWorx to create and initialize the entities.

Defining the Extension

The `metadata.xml` file must contain only one `<ExtensionPackage>` element that provides details of the extension.

For example:

```
<Entities>
 <ExtensionPackages>
```

```
  <ExtensionPackage
   name="MyExtension"
   packageVersion="1.0.0"
   vendor="ThingWorx - A PTC Business"
   description="My Extension description"
   minimumThingWorxVersion="7.1.0"
   dependsOn="ExtensionOne:2.5.1,ExtensionTwo:1.2.0">
  </ExtensionPackage>
 </ExtensionPackages>
</Entities>
```

Tip

If you are using the Eclipse Plugin, the **New ThingWorx Project** action will generate a new `metadata.xml` file and fill in the details. For more information, see Creating an Extension Project.

<ExtensionPackage> Attribute	Description
name	The name of the extension. This is a required attribute whose value must follow the naming convention for ThingWorx entities.
packageVersion	The version of the extension. This is a required attribute whose value must follow the `<major>.<minor>.<patch>` format, where each part of the version is numeric. Extensions should follow the semantic versioning rules outlined at http://www.semver.org.
vendor	The name of the organization that created the extension. This is an optional attribute.
description	Information about what the extension does. This is an optional attribute that is displayed in the **Manage Extensions** screen of Composer.
minimumThingWorxVersion	Specifies the minimum version of ThingWorx with which the extension is compatible. This is a required attribute whose value must follow the `<major>.<minor>.<patch>` format, where each part of the version is numeric. If you try to import the extension into a version of ThingWorx that is older than what is specified in this attribute, it will fail.

<ExtensionPackage> Attribute	Description
dependsOn	A list of extensions on which the extension depends. This is an optional attribute. When specified, its value must be a comma-separated list of extension names and versions in the `<name>:<major>.<minor>.<patch>` format, where each part of the version is numeric. For example, *dependsOn= "ExtensionOne:2.5.1,ExtensionTwo:1.2.0"*. If you try to import the extension into an instance of ThingWorx that does not have these extensions installed, it will fail.
migratorClass	The fully-qualified Java class name of the migrator that should be run when importing data from prior versions of the extension. This is an optional attribute. For more information, see Adding an Extension Migrator. **Note** This attribute is supported in ThingWorx 7.1 and higher.

In addition to defining this extension information, the `metadata.xml` file also contains information about other artifacts included in the extension.

Defining Thing Templates

Note

For details about the ThingWorx model, see the ThingWorx Model Definition and Composer topic in the ThingWorx Help Center.

Each Java-based thing template must have two entries in the `metadata.xml` file:

- A `<ThingPackage>` element under the `<Entities><ThingPackages>` element, which specifies the class name of the thing template
- A `<ThingTemplate>` element under the `<Entities><ThingTemplates>` element, which refers to the name of the thing package

For example:

```
<Entities>
 <ThingPackages>
  <ThingPackage name="MyThingPackage"
   description="A description"
   className="com.acmecorp.things.MyThingTemplate" />
 </ThingPackages>
 <ThingTemplates>
  <ThingTemplate name="MyThingTemplate"
   description="My Thing Template description"
   thingPackage="MyThingPackage">
 </ThingTemplates>
</Entities>
```

Tip

If you are using the Eclipse Plugin, the **New Thing Template** action will generate the necessary elements in the `metadata.xml` file and fill in the details. For more information, see Creating Entities.

Note

If a Java-based extension defines a thing template as Java classes using annotations but they are not included in the entity's `metadata.xml` file, ThingWorx cannot detect if the entity's base template changed and the import of the extension will fail.

<ThingPackage> Attribute	Description
name	The name of the thing package. This is a required attribute whose value must follow the naming convention for ThingWorx entities.
description	Provides information about the thing package. This is an optional attribute.
className	The fully-qualified Java class name of the `ThingTemplate` class that corresponds to the thing package. This is a required attribute.

<ThingTemplate> Attribute	Description
name	The name of the thing template. This is a required attribute whose value must follow the naming convention for ThingWorx entities. This is the name that will be displayed in Composer in the list of thing templates.
description	Provides information about the thing package. This is an optional attribute.
thingPackage	The name of the thing package that corresponds to the thing template. It must match the `ThingPackage` name attribute. This is a required attribute.

Note

Setting the `aspect.isEditableExtensionObject` attribute on a Thing Template is not supported.

Defining Thing Shapes

Each Java-based thing shape must have a `<ThingShape>` element under the `<Entities><ThingShapes>` element, which specifies the class name of the thing shape.

For example:

```
<Entities>
 <ThingShapes>
  <ThingShape name="MyThingShape"
   description="My Thing Shape description"
   className="com.acmecorp.things.MyThingShape"/>
 </ThingShapes>
</Entities>
```

Tip

If you are using the Eclipse Plugin, the **New Thing Shape** action will generate the necessary elements in the `metadata.xml` file and fill in the details. For more information, see Creating Entities.

<ThingShape> Attribute	Description
name	The name of the thing shape. This is a required attribute whose value must follow the naming convention for ThingWorx entities.
description	Provides information about the thing shape. This is an optional attribute.
className	The fully-qualified Java class name of the `ThingShape` class. This is a required attribute.
aspect.isEditableExtensionObject	Indicates if the entity can be edited in ThingWorx Composer. This is an optional attribute and defaults to false. **Note** This attribute is supported in ThingWorx 7.0 and higher.

Defining Resources

Each Java-based resource must have a `<Resource>` element under the `<Entities><Resources>` element, which specifies the class name of the resource.

For example:

```
<Entities>
 <Resources>
  <Resource name="MyResource"
   description="My Resource description"
   className="com.acmecorp.resources.MyResource" />
 </Resources>
</Entities>
```

Tip

If you are using the Eclipse Plugin, the **New Resource** action will generate the necessary elements in the `metadata.xml` file and fill in the details. For more information, see Creating Entities.

<Resource> Attribute	Description
name	The name of the resource. This is a required attribute whose value must follow the naming convention for ThingWorx entities.
description	Provides information about the resource. This is an optional attribute.
className	The fully-qualified Java class name of the `Resource` class. This is a required attribute.
aspect.isEditableExtensionObject	Indicates if the entity can be edited in ThingWorx Composer. This is an optional attribute and defaults to false. **Note** This attribute is supported in ThingWorx 7.0 and higher.

Defining Authenticators

Each Java-based authenticator must have an `<Authenticator>` element under the `<Entities><Authenticators>` element, which specifies the class name of the authenticator.

For example:

```
<Entities>
 <Authenticators>
  <Authenticator name="MyAuthenticator"
   description="My Authenticator description"
   className="com.acmecorp.authenticators.MyAuthenticator"
   aspect.isEditableExtensionObject=”true” />
 </Authenticators>
</Entities>
```

Tip

If you are using the Eclipse Plugin, the **New Authenticator** action will generate the necessary elements in the `metadata.xml` file and fill in the details. For more information, see Creating Entities.

<Authenticator> Attribute	Description
name	The name of the authenticator. This is a required attribute whose value must follow the naming convention for ThingWorx entities.
description	Provides information about the authenticator. This is an optional attribute.
className	The fully-qualified Java class name of the `Authenticator` class. This is a required attribute.
aspect.isEditableExtensionObject	Indicates if the entity can be edited in ThingWorx Composer after import. This should be set to true so it is enabled in Composer after import.

Tip

For more information about deploying and enabling authenticators after import into ThingWorx, see the *Authenticator Sample Extension Configuration* topic in the ThingWorx Help Center.

Defining Directory Services

Each Java-based directory service must have a `<DirectoryService>` element under the `<Entities><DirectoryServices>` element, which specifies the class name of the directory service.

For example:

```
<Entities>
 <DirectoryServices>
  <DirectoryService name="MyDirectoryService"
   description="My Directory Service description"
   className="com.acmecorp.resources.MyDirectoryService"
   aspect.isEditableExtensionObject=”true” />
 </DirectoryServices>
</Entities>
```

Tip

If you are using the Eclipse Plugin, the **New Directory Service** action will generate the necessary elements in the `metadata.xml` file and fill in the details. For more information, see Creating Entities.

<Directory Service> Attribute	Description
name	The name of the directory service. This is a required attribute whose value must follow the naming convention for ThingWorx entities.
description	Provides information about the directory service. This is an optional attribute.
className	The fully-qualified Java class name of the `DirectoryService` class. This is a required attribute.
aspect.isEditableExtensionObject	Indicates if the entity can be edited in ThingWorx Composer after import. This should be set to true so the entity can be configured in Composer after import.

Tip

For more information about creating a directory service, see the *Directory Services Authentication* topic in the ThingWorx Help Center.

Defining Script Function Libraries

Each Java-based script function library must have a `<ScriptFunctionLibrary>` element under the `<Entities><ScriptFunctionLibraries>` element, which specifies the class name of the script function library. For each function in the library, `<FunctionDefinition>` elements must be added to specify the function's name, parameters, and return type.

For example:

```
<Entities>
 <ScriptFunctionLibraries>
```

```
<ScriptFunctionLibrary name="MyScriptFunctionLibrary"
 description="My Script Function Library description"
 className=" com.acmecorp.sfl.MyScriptFunctionLibrary">
 <FunctionDefinitions>
  <FunctionDefinition name="calculateIt"
   description="Performs a calculation">
   <ParameterDefinitions>
    <FieldDefinition name="parm1"
     description="The first parameter"
     baseType="INTEGER"/>
    <FieldDefinition name="parm2"
     description="The second parameter"
     baseType="INTEGER"/>
   </ParameterDefinitions>
   <ResultType name="resultingValue" baseType="INTEGER"
    description="The return type"/>
  </FunctionDefinition>
 </FunctionDefinitions>
</ScriptFunctionLibrary>
</ScriptFunctionLibraries>
</Entities>
```

Tip

If you are using the Eclipse Plugin, the **New Script Function** action will generate the necessary elements in the `metadata.xml` file and fill in the details. Function definition elements must be added manually. For more information, see Creating Entities.

<ScriptFunctionLibrary> Attribute	Description
name	The name of the script function library. This is a required attribute whose value must follow the naming convention for ThingWorx entities.
description	Provides information about the script function library. This is an optional attribute.
className	The fully-qualified Java class name of the `ScriptFunctionLibrary` class. This is a required attribute.

<FunctionDefinition> Attribute	Description
name	The name of the function. This is a required attribute whose value must match the name of a corresponding `public static` method in the JavaScript Function Library class.
description	Provides information about the function. This description is displayed in ThingWorx Composer.

<FieldDefinition> Attribute	Description
name	The name of the field. This is a required attribute.
description	Provides information about the parameter. This description is displayed in ThingWorx Composer.
baseType	The base type of the field. This is a required attribute whose value must be a valid ThingWorx base type, such as *STRING* or *NUMBER*. When you create a property in Composer, you can see the full list in the **Base Type** dropdown. For more information, see com.thingworx.types.BaseTypes in the *ThingWorx Platform API Documentation* topic in the ThingWorx Help Center. **Note** The value must be uppercase.

<ResultType> Attribute	Description
name	The name of the return value. This is a required attribute.
description	Provides information about the return value. This description is displayed in ThingWorx Composer.
baseType	The base type of the return value of the method. This is a required attribute whose value must be a valid ThingWorx base type, such as *STRING* or *NUMBER*. When you create a property in Composer, you can see the full list in the **Base Type** dropdown. For more information, see com.thingworx.types.BaseTypes in the *ThingWorx Platform API Documentation* topic in the ThingWorx Help Center. **Note** The value must be uppercase.

Defining JAR Resources

For each JAR file included in the extension, an entry must be added in the metadata.xml file. The `<JarResources>` element contains a reference to the JAR that contains the extension's Java-based entity classes. It is used to reference third-party JAR files on which the extension depends.

For example:

```
<ExtensionPackages>
 <ExtensionPackage ...>
  <JarResources>
   <FileResource type="JAR" file="MyExtension.jar"
    description="Contains the extension's custom classes" />
   <FileResource type="JAR" file="ThirdPartyLib.jar
    description="My Extension depends on this JAR" />
  </JarResources>
 </ExtensionPackage>
</ExtensionPackages>
```

Tip

If you are using the Eclipse Plugin, the **New Jar Resource** action will generate the necessary elements in the `metadata.xml` file and fill in the details. For more information, see Creating Entities.

Caution

Do not include third-party JAR files in your extension that are already included in the ThingWorx Platform.

Entities Created in ThingWorx Composer

You can create entities in ThingWorx Composer for inclusion in the extension (mashups) and export them using the **Export to File (XML Format)** or **Export Source Control Entities** actions. An extension developer can include the XML files in the extension zip file under the **Entities** folder.

Tip

Tag your extension entities during development so it's easier to export them together.

Each XML file should be placed in the folder structure that mirrors the element hierarchy in the XML file itself. For example, a thing shape is defined in the XML file in the element hierarchy `<Entities>` `<ThingShapes>` `<ThingShape>`, so the XML file should be placed in the `Entities\ ThingShapes` folder in the extension.

No changes to the metadata.xml file are needed when including Composer-created entities in an extension.

An example of the `Entities` folder structure is as follows:

```
/Entities
 /Mashups
  /MyMashup.xml
 /ThingShapes
  /MyThingShape1.xml
  /MyThingShape2.xml
 /ThingTemplates
  /MyThingTemplate.xml
```

Tip

The easiest way to add Composer-created entities is to tag entities created for the extension and export them using the **Export Source Control Entities** action.

In Composer, use `Import/Export` > `Export` > `Source Control Entities` to export your extension entities to an `/Entities` folder at the root of the extension package structure. The resulting `Entities` folder under `ThingworxStorage/repository/<RepositoryName>` can be placed directly into the extension zip file.

Tip

If you are using the Eclipse Plugin, the **Import ThingWorx Entities** action places the files in the correct location in the project. For more information, see Importing Composer-Created Entities.

Java-Based Entities

Java-based entities created in an extension are basically the same as those created in ThingWorx Composer. They can also define services, properties, events, configuration parameters, and so on.

The main difference between the two types of entities is that Composer-created entities use JavaScript for services, and their source code is visible in Composer. Java-based entities use Java for services, and the source code is not visible in Composer. Java-based entities can also define configuration values and lifecycle behavior.

Tip

Writing services in Java provides several advantages, including the ability to write unit tests and to more easily perform remote debugging with breakpoints.

ThingWorx Extension SDK

Java-based entities created in extensions must use the ThingWorx Extension SDK to access supported ThingWorx Platform classes and APIs. The SDK exposes many built-in services that make manipulating Platform objects easier. It also includes Javadoc documentation to aid in the development process.

API Compatibility

Only service APIs (consumable from a REST API) are supported in ThingWorx 6.5 and prior releases.

Java classes and methods may not be backward compatible from ThingWorx 6.6 and future releases.

New APIs may be added in future releases. All supported APIs will be included in the Javadoc. Their behavior should not change between releases. If the behavior of an API changes, it will be deprecated and an alternative pattern will be provided in the Javadoc.

Creating Thing Templates

To create a thing template, create a Java class that extends com.thingworx.thing.Thing and update the metadata.xml file. Each thing template needs the following entries in the metadata.xml file:

- A <ThingPackage> element
- A <ThingTemplate> element

For more information, see metadata.xml file.

Note

You should create thing templates that derive from extension thing templates and implement extension thing shapes. You can add custom properties and services to your extension things, thing shapes, and thing templates.

To define a base thing template, add the following annotation to the Java class:

```
@ThingworxBaseTemplateDefinition(name="RemoteThing")
```

If not specified, the base thing template defaults to *GenericThing*.

To define implemented thing shapes, add the following annotations to the Java class:

```
@ThingworxImplementedShapeDefinitions(shapes = {
  @ThingworxImplementedShapeDefinition(name = "ExampleShape1"),
  @ThingworxImplementedShapeDefinition(name = "ExampleShape2")
})
```

Once the class is created, you can add services, properties, configuration tables, events, and subscriptions to the thing template. For more information, see Adding Services, Properties, and Other Methods to an Entity.

Tip

If you are using the Eclipse Plugin, the **New Thing Template** action will generate the Java source file with the appropriate annotations and automatically update the metadata.xml file. For more information, see Creating Entities.

You can override the following methods from the thing superclass to add custom functionality in the thing template (see the Javadoc documentation for the com.thingworx.things.Thing class for more information):

- `protected void cleanupThing() throws Exception;`
- `protected void initializeThing() throws Exception;`
- `protected void preprocessSetPropertyVTQ(ThingProperty, VTQ, boolean) throws Exception;`
- `protected void processStartNotification();`
- `protected void startThing() throws Exception;`
- `protected void stopThing() throws Exception;`

For example, you can override the `initializeThing()` method. This method is called when the thing is created or saved. It is used to set up any functionality needed for your Java code to function properly. It may be used to create a data table and associate it with your thing when the thing is created. This is accomplished by checking for the existence of the data table before creating it, since the same method is called when the thing is saved.

Creating Thing Shapes

To create a thing shape, create a Java class that extends `java.lang.Object` (it does not need to extend any ThingWorx classes) and update the metadata.xml file. Each thing shape must have a `<ThingShape>` element. For more information, see metadata.xml File.

Once the class is created, you can add services, properties, events, and subscriptions to the thing shape. For more information, see Adding Services, Properties, and Other Methods to Entities.

Tip

If you are using the Eclipse Plugin, the **New Thing Shape** action will generate the Java source file with the appropriate annotations and automatically update the metadata.xml file. For more information, see Creating Entities.

Creating Resources

Resources are one of the simplest parts of an extension to create but can offer very powerful functionality. They are designed to expose services with no stateful behavior. You can call the service from anywhere, but you need to provide data that could vary in different scenarios. No properties, configurations, or events are associated with a resource. Resources are useful for things like encoding, formatting, and complex math equations.

Best Practice

When to create a resource or helper thing:

- A resource is little easier to assemble and does not require a thing template. It may be beneficial if you need to expose services without a state-based behavior. Note that resources cannot be created in Composer.
- A helper thing is a better choice if you need to include properties, configuration, or events. Things and thing templates can be created in Composer.

For more information about creating resources, things, and thing templates, see the ThingWorx Help Center.

To create a resource, create a Java class that extends `com.thingworx.resources.Resource` and update the metadata.xml file. Each resource must have a `<Resource>` element. For more information, see metadata.xml file.

Once the class is created, services can be added to the resource (since resources are stateless, do not add properties, configuration tables, subscriptions, and events). For more information, see Adding Services, Properties, and Other Methods to Entities.

Tip

If you are using the Eclipse Plugin, the **New Resource** action will generate the Java source file with the appropriate annotations and automatically update the metadata.xml file. For more information, see Creating Entities.

Creating Authenticators

To create an authenticator, create a Java class that extends `com.thingworx.security.authentication.CustomAuthenticator` and update the metadata.xml file. Each authenticator must have an `<Authenticator>` element in the metadata.xml file. For more information, see metadata.xml File.

The following methods need to be implemented. For more information, see the Javadoc for the `com.thingworx.security.authentication.IAuthenticator` interface and the *Authenticator Sample Extension Configuration* topic in the ThingWorx Help Center.

- `public void authenticate(HttpServletRequest, HttpServletResponse) throws AuthenticatorException;`
- `public void issueAuthenticationChallenge(HttpServletRequest, HttpServletResponse) throws AuthenticatorException;`
- `public boolean matchesAuthRequest(HttpServletRequest) throws AuthenticatorException;`

Note

The parameter classes and exceptions used in these methods are part of the javax.servlet third-party JAR, which must be in the build path for the custom authenticator to compile. However, this JAR is a compile-time dependency only and must not be added as part of the extension itself.

Tip

If you are using the Eclipse Plugin, the **New Authenticator** action will generate the Java source file with the appropriate annotations and automatically update the metadata.xml file. It will also prompt for the location of the Web server to automatically update the build path with the javax.servlet JAR. For more information, see Creating Entities.

Creating Directory Services

To create a directory service, create a Java class that extends `com.thingworx.security.directoryservices.DirectoryService` and update the metadata.xml file. Each directory service must have a `<DirectoryService>` element in the metadata.xml file. For more information, see metadata.xml File.

The following methods must be implemented (see Javadoc documentation for the `com.thingworx.security.directoryservices.DirectoryService` interface):

- `public void ValidateCredentials(User user, String password) throws Exception;`
- `public void ValidateCredentials(User user, String password, HttpServletRequest req) throws Exception;`

Note

The *HttpServletRequest* parameter class used in these methods is part of the `javax.servlet` third-party JAR which must be in the build path for the custom directory service to compile. However, this JAR is a compile-time dependency only and must **not** be added as part of the extension itself.

Tip

If you are using the Eclipse Plugin, the **New Directory Service** action will generate the Java source file with the appropriate annotations and automatically update the metadata.xml file. For more information, see Creating Entities.

Tip

For more information about creating directory services, see the *Directory Services Authentication* topic in the ThingWorx Help Center.

Creating Script Function Libraries

To create a script function library, create a Java class that extends `java.lang.Object` and update the metadata.xml file.

In the Java class, define public static methods that provide the desired functionality. Each method must have a signature of the form:

```
public static Object doSomething(
 org.mozilla.javascript.Context cx,
 org.mozilla.javascript.Scriptable thisObj,
 Object[] args,
 org.mozilla.javascript.Function funObj) throws Exception {

 // do something and return something
}
```

The `org.mozilla.javascript` classes are part of the Rhino JavaScript library, which must be part of the extension's build path.

Each script function library must have a `<ScriptFunctionLibrary>` element in the metadata.xml file and a corresponding `<FunctionDefinition>` element for each method with the appropriate sub-elements defining the parameters and return type. For more information, see metadata.xml File.

Tip

If you are using the Eclipse Plugin, the **New Script Function Library** action will generate the Java source file with the appropriate annotations and automatically update the metadata.xml file. For more information, see Creating Entities.

Adding Services, Properties, and Other Methods to Entities

One of the main tasks when creating an extension is to add custom behavior to entities by defining services, properties, events, and/or subscriptions on them.

Services

ThingWorx Platform services are chunks of code that execute specific functionality and are exposed as REST API calls. They are used internally in the Platform, exposed to be used by mashups, and can be reached from any external source with the proper credentials. In Java code, annotations are used to mark a Java method as the entry point for a service.

Although methods are used to define services on the ThingWorx Platform, all methods are not automatically considered services. There can be many internal methods inside your Java code that add functionality to the extension, but only the methods marked with the proper annotations will be exposed as a service on the Platform.

Tip

If you are using the Eclipse Plugin, the **Add Service** action will add the `Service` method and the necessary annotations described below to the Java source file. For more information, see Adding Services, Properties, Configuration Tables, Subscriptions, and Events.

Creating Services

When you are creating custom services in your extension, you should decide which users or user groups should have the ability to invoke this service. For more information about visibility and permissions, see the ThingWorx Help Center.

Defining Services

Services are defined by adding annotations to the method of a class. The annotation identifies that this method should be treated as a ThingWorx service and defines the inputs and outputs of the service. The following annotations are part of a service definition:

- `@ThingworxServiceDefinition`
- `@ThingworxServiceResult`
- `@ThingworxServiceParameters`

The @ThingworxServiceDefinition Annotation

The `@ThingworxServiceDefinition` annotation defines the name and description of a service and associates that service with a given method. It is placed above a method definition.

The most commonly used attributes of this annotation are listed below. For a complete set of attributes, see the Javadoc documentation for the `com.thingworx.metadata.annotations.ThingworxServiceDefinition` class.

@ThingworxServiceDefinition Attribute	Description
name	The name of the service. It is standard convention for service names to start with a capital letter. The name of the service must be identical to the name of the method. This is a required attribute.
description	A short description for the service. The description should provide information on functionality of the service and other pertinent information for users. This description will appear in a tool tip when hovering over the service name in ThingWorx Composer.
category	A category that conceptually groups related services. This is an optional attribute.

The @ThingworxServiceResult Annotation

The `@ThingworxServiceResult` annotation defines the output type of the service. This base type is treated the same as property definitions and is case-sensitive. For some base types, such as **INFOTABLE** or **THINGNAME**, you may need to add aspects to define more complex types.

The most commonly used attributes of this annotation are listed below. For a complete set of attributes, see the Javadoc documentation for the `com.thingworx.metadata.annotations.ThingworxServiceResult` class.

@ThingworxServiceResult Attribute	Description
name	The name of the result as referenced by other services when invoked via the REST API or the `processServiceRequest()` methods. If the base type of the service is **INFOTABLE**, the name is ignored. Otherwise, the result of the service is an info table with one column whose key is equal to the name parameter. This is a required attribute.
description	A short description for the service result. This description appears when viewing the service details in Composer.
baseType	The base type for the result. When defining the base type for the result, the matching Java type must be used for the method's result. This is a required attribute whose value must be a valid ThingWorx type such as STRING or NUMBER. When creating a property in Composer, you can see the full list in the **Base Type** dropdown list. See `com.thingworx.types.BaseTypes` in the *ThingWorx Platform API Documentation* topic of the ThingWorx Help Center. **Note** The value must be uppercase.

In the following code snippet, aspects were needed to define the data shape of an info table. If the base type is STRING, these aspects would not be needed. This example is showing the service definition entered above a method:

```
// Access a Resource
@ThingworxServiceDefinition( name="AccessAResource",
 description="Execute a service of a resource" )
@ThingworxServiceResult( name="result",
 description="Matching entries", baseType= "INFOTABLE",
 aspects={"dataShape:RootEntityList"} )
public InfoTable AccessAResource() throws Exception {
}
```

The @ThingworxServiceParameters Annotation

ThingworxServiceParameters are used to associate ThingWorx inputs with method input parameters. This defines what input types will be available for binding in ThingWorx Platform. The annotations should be entered before the parameter definition, inside the method's declaration.

For example, the following code snippet shows service parameters added to a method's input parameters:

```
// Property set of another Thing
@ThingworxServiceDefinition( name="SetHelloWorldProperty",
 description="Set a property on Hello World" )
public void SetHelloWorldProperty(
 @ThingworxServiceParameter( name="thingToSet",
  description="The thing you're setting", baseType="THINGNAME",
  aspects={"thingTemplate:HelloWorld"} ) String thingToSet,
 @ThingworxServiceParameter( name="numberPropertyToSet",
  description="The property name you're setting",
  baseType="STRING" ) String numberPropertyToSet,
 @ThingworxServiceParameter( name="numberValueToSet",
  description="The property value you're setting",
  baseType="NUMBER") Double numberValueToSet) throws Exception {

  Thing helloThing = ThingUtilities.findThing(thingToSet);
  helloThing.setPropertyValue(numberPropertyToSet,
   new NumberPrimitive(numberValueToSet));
}
```

For more information, see the Javadoc documentation for `com.thingworx.metadata.annotations.ThingworxServiceParameter`.

Properties

Properties are defined using the `@ThingworxPropertyDefinitions` annotation. This annotation specifies a list of properties associated with the thing template or thing shape that is being defined. Each property uses its own annotation to define its `name`, `description`, `baseType`, and `aspects`. See the HelloWorldThing example.

> **Note**
>
> If you are using the Eclipse Plugin, the **Add Property** action will add the necessary property annotations described below to the Java source file. See Adding Services, Properties, Configuration Tables, Subscriptions, and Events for more information.

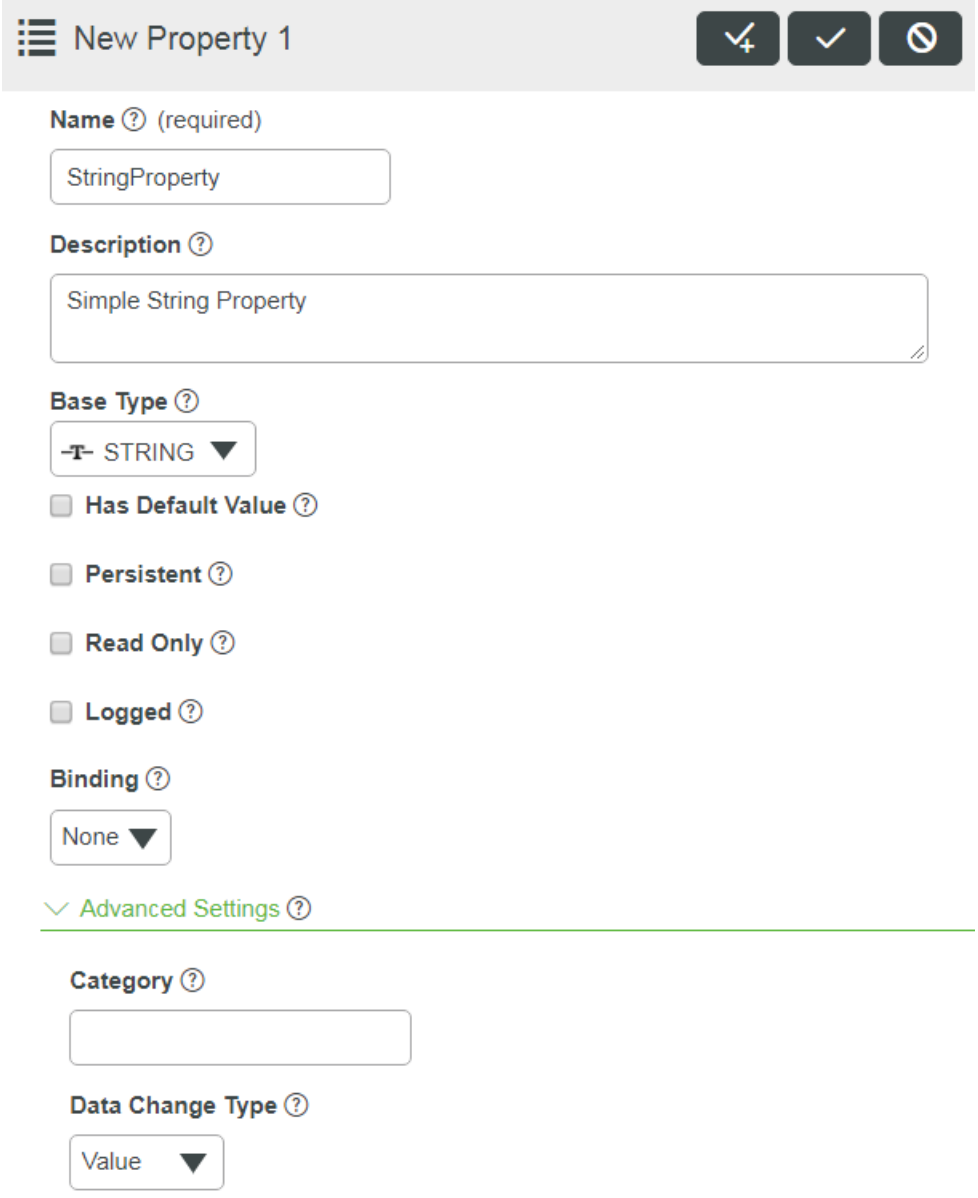
@ThingworxPropertyDefinition Attribute	Description
`name`	The name of the property
`description`	A string with a unique description for this property
`baseType`	The base type of the property. The base type of the field. This is a required attribute whose value must be a valid ThingWorx type such as STRING or NUMBER. When creating a property in Composer, you can see the full list in the **Base Type** dropdown. See com.thingworx.types.BaseTypes in the *ThingWorx Platform API Documentation* topic of the ThingWorx Help Center. **Note** The value must be uppercase.
`aspects`	Metadata for the given property. This parameter defines various settings for the property in the form of key-value pairs, such as *minValue* or *isPersistent*. All aspects are defined using camel case. You can find the full list of aspects you can set by viewing the property creation window in Composer. Examples: "isPersistent:true" or "dataShape:myDataShape" or "thingTemplate:myTemplate" **Tip** Complex data types like INFOTABLE require extra details associated with the type. These are defined in the aspects section using camel case for the name.

For example, the following code snippet shows property definitions above the class definition:

```
@ThingworxPropertyDefinitions(properties = {
 @ThingworxPropertyDefinition(name="StringProperty1",
```

```
  description="Sample string property", baseType="STRING",
  aspects={"isPersistent:false","isReadOnly:false"}),
 @ThingworxPropertyDefinition(name="NumberProperty1",
  description="Sample number property", baseType="NUMBER",
  aspects={"isPersistent:false","isReadOnly:false"}),
})
public class HelloWorldThing extends Thing {
}
```

The image below shows the fields available when you are adding a new property in ThingWorx Composer:



For more information, see Javadoc for `com.thingworx.metadata.annotations.ThingworxPropertyDefinition`.

Configuration Tables

Configuration tables are used for things, thing templates, thing shapes, and mashups to store values, like property values that do not change often. The most common use of configuration tables is to store credentials and host information for an external resource. These are defined in a similar way to properties.

To use configurations, you define a configuration table with a name and description and indicate if it can store multiple rows. Similar to properties, some configuration values have aspects associated with them as well.

You can define a configuration table by using one of the following:

- Java annotations on page 38
- XML import/export on page 39
- REST API on page 40

Adding Annotations on Java Classes

You can create configuration tables by adding annotations on Java classes. For more information, see Javadoc documentation for `com.thingworx.metadata.annotations.ThingworxConfigurationTableDefinition`.

Tip

If you are using the Eclipse Plugin, the **Add Configuration Table** action will add the annotations described below to the Java source file. For more information, see Adding Services, Properties, Configuration Tables, Subscriptions, and Events.

For example, this code snippet shows the configuration table definitions located above the class definition:

```
@ThingworxConfigurationTableDefinitions(tables = {
 @ThingworxConfigurationTableDefinition(
  name="ConfigTableExample1",
  description="Example 1 config table", isMultiRow=false,
  dataShape = @ThingworxDataShapeDefinition( fields = {
   @ThingworxFieldDefinition(name="field1",
    description="",baseType="STRING"),
   @ThingworxFieldDefinition(name="field2",
    description="",baseType="NUMBER"),
   @ThingworxFieldDefinition(name="field3",
```

```
    description="",baseType="BOOLEAN"),
   @ThingworxFieldDefinition(name="field4",
    description="",baseType="USERNAME"),
 } ) ),
 @ThingworxConfigurationTableDefinition(
  name="ConfigTableExample2",
  description="Example 2 config table", isMultiRow=true,
  dataShape = @ThingworxDataShapeDefinition( fields = {
   @ThingworxFieldDefinition(name="columnA",
    description="",baseType="STRING"),
   @ThingworxFieldDefinition(name="columnB",
    description="",baseType="NUMBER"),
   @ThingworxFieldDefinition(name="columnC",
    description="",baseType="BOOLEAN"),
   @ThingworxFieldDefinition(name="columnD",
    description="",baseType="USERNAME"),
 } ) )
})
public class GoodByeThing extends Thing {
}
```

XML Import/Export

You can also define configuration tables on things, thing templates, thing shapes, and mashups through XML imports. The following is a configuration table definition for an entity exported as XML:

```
<ConfigurationTableDefinitions>
<ConfigurationTableDefinition category="TemplateConfigTables"
dataShapeName="MyDS" description="Template Config Table" isHidden=
"false"
isMultiRow="true" name="ConfigTableOnTemplate" ordinal="2" source=
"REST"/>
</ConfigurationTableDefinitions>
```

Note

Configuration table defintions created by annotations do not appear in exported XML.

The following is an example of configuration table data exported as XML:

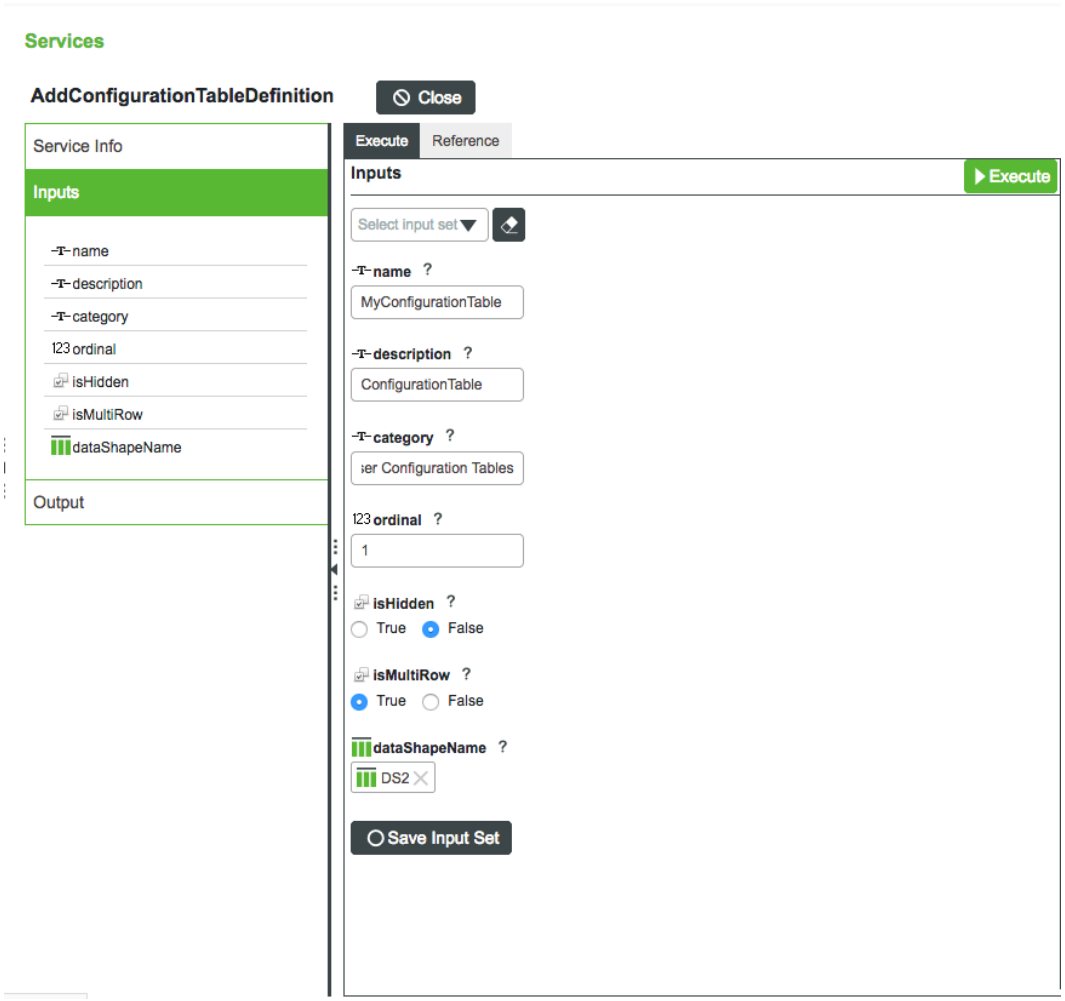
```
<ConfigurationTables>
<ConfigurationTable description="Template Config Table"
isMultiRow="true" name="ConfigTableOnTemplate" ordinal="2">
<DataShape>
<FieldDefinitions>
```

```
<FieldDefinition aspect.isPrimaryKey="true" aspect.tagType=
"ModelTags" baseType="STRING" description="" name="p1" ordinal="1"/>
<FieldDefinition aspect.isPrimaryKey="false" aspect.tagType=
"ModelTags" baseType="STRING" description="" name="p2" ordinal="2"/>
<FieldDefinition aspect.isPrimaryKey="false" aspect.tagType=
"ModelTags" baseType="STRING" description="" name="p3" ordinal="3"/>
</FieldDefinitions>
</DataShape>
<Rows>
<Row>
<p1>
<![CDATA[1]]>
</p1>
<p2>
<![CDATA[2]]>
</p2>
<p3>
<![CDATA[3]]>
</p3>
</Row>
</Rows>
</ConfigurationTable>
</ConfigurationTables>
```

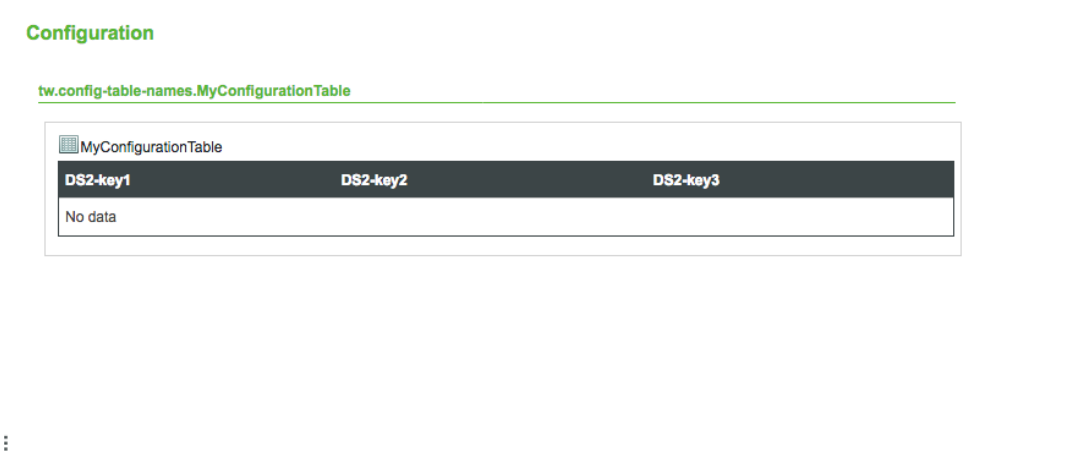
Adding Configuration Tables Using the REST API (for Things and Mashups)

You can add configuration tables to entities such as things or mashups using REST API services. Every configuration table is associated with a data shape. The data shape for a multi-row configuration table must have a primary key. Once the configuration table is saved, its data shape cannot be changed; however, you can make changes on the data shape fields. You can add or delete a field from the data shape or change a field's base type. Changing the base type could impact the data on the configuration table. For example, changing a *String* base type to *Integer* could result in loss of configuration table data.

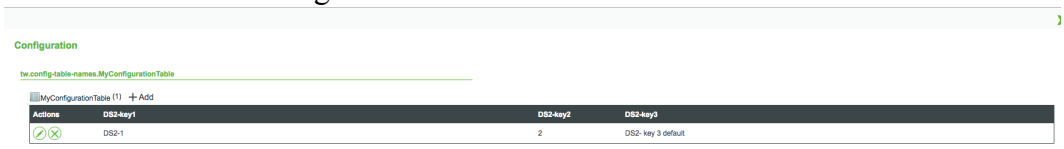
The following image shows a configuration table definition using services on a thing in ThingWorx Composer:



The configuration table created above appears under the **Configuration** tab for the thing in ThingWorx Composer as follows:

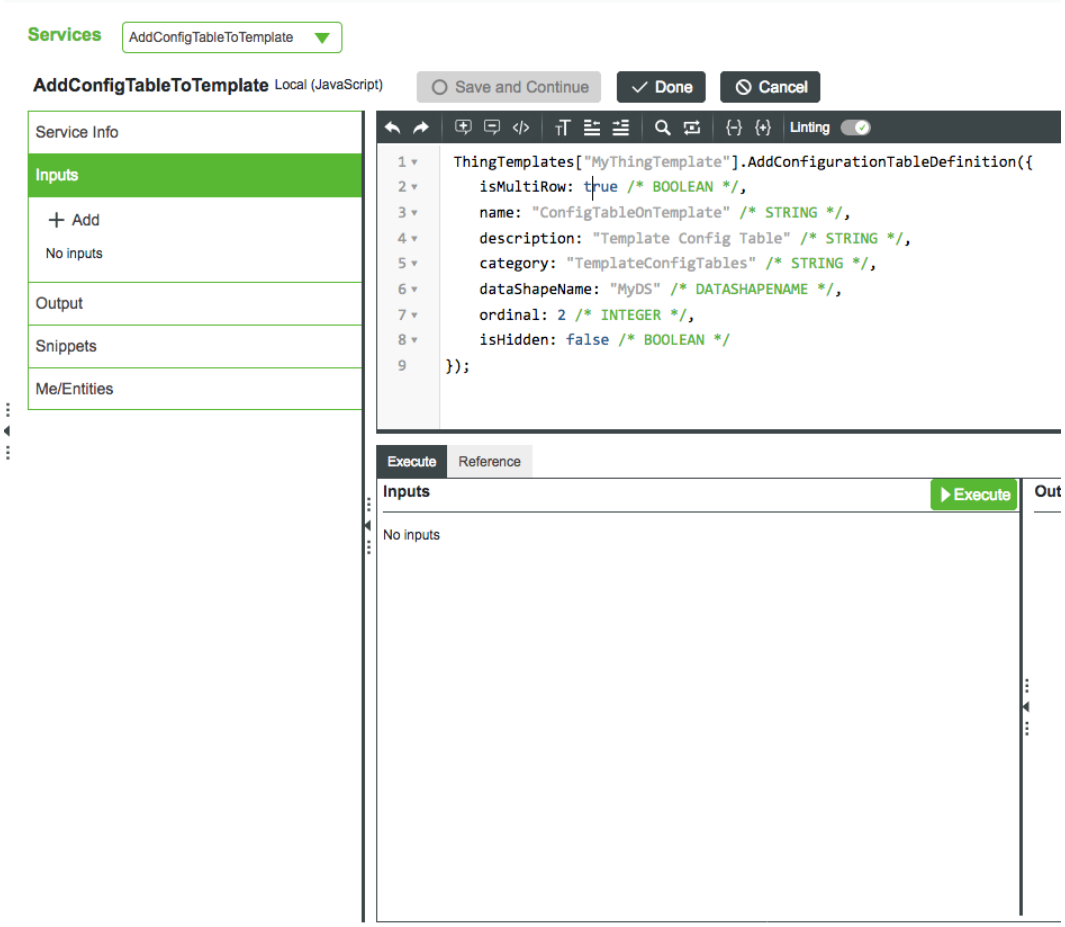


To add data to the configuration table, open the entity in edit mode and select the **Add** button next to configuration table name.

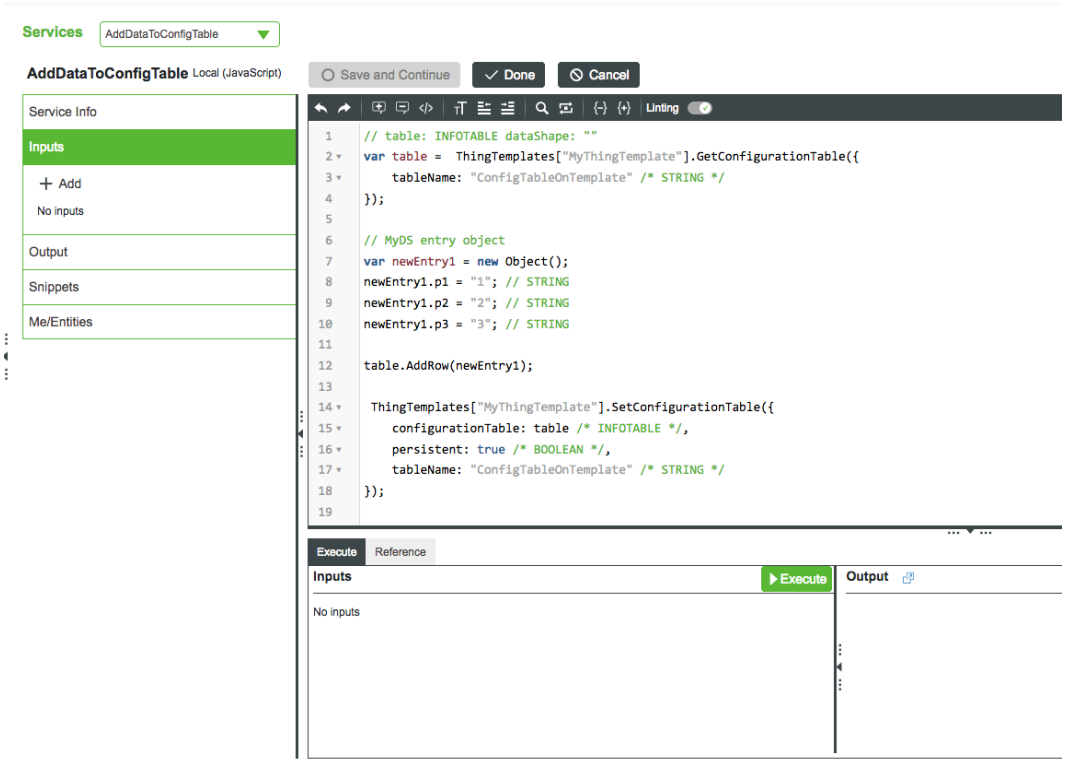


Adding Configuration Tables Using the REST API (for Thing Templates and Thing Shapes)

To add configuration tables to entities such as a thing template or thing shape, you can write a service on a thing that invokes the **AddConfigurationTableDefinition** REST API on the thing template. Below is an example of a user-defined service (**AddConfigTableToTemplate**) that adds a configuration table to a thing template (`MyThingTemplate`). Similarly, you can write a service to add a configuration table to a thing shape.



You can add data to the configuration table from ThingWorx Composer or with a service like the following:



Configuration Table and Configuration Data Inheritance

Configuration tables and their data that are created on thing template and thing shape entities are inherited by implementing a thing entity. Any definition changes made to the configuration tables at the template or shape level are reflected on implementing things that inherited the tables from their template or shape.

A new thing entity will inherit configuration data from its shapes or template. However, after the thing is created, it will not take later data changes to its inherited tables from its template or shapes.

For example:

- `ThingTemplateA` has `configTable1`
- `ThingShapeA` has `configTable2`

`ThingA` is created, which is based on `ThingTemplateA` and implements `ThingShapeA`.

`ThingA` will inherit `configTable1` and `configTable2` with data from its template and shape respectively.

If `ThingTemplateA` changes `configTable1` from *singleRow* to *multiRow*, then `ThingA` will reflect that change.

But if `ThingTemplateA` changes the data in `configTable1`, then `ThingA` will not inherit the change in data. `ThingA` will keep whatever data it has for `configTable1`. `ThingA` can also change the data on `configTable1` and `configTable2` and the change in data will not cascade to its template or shape or other things which have inherited this configuration table.

Events

You can define custom events in your extension that occur when certain conditions are met. These events can be subscribed to by other entities using the same mechanisms for built-in events. Events can trigger custom functionality. They require a predefined data shape. The data shape stores data associated with the event, which can be accessed by a subscription.

For more information, see the Javadoc documentation for `com.thingworx.metadata.annotations.ThingworxEventDefinition`.

Tip

If you are using the Eclipse Plugin, the **Add Event** action will add the necessary annotations described below to the Java source file. For more information, see Adding Services, Properties, Configuration Tables, Subscriptions, and Events.

Note

If the data shape is custom, it must be included in the `Entities` folder to be recognized on import. For more information, see Entities Created in Composer.

For example, the following code snippet defines an event:

```
@ThingworxEventDefinitions(events = {
@ThingworxEventDefinition(name="SalutationSent",
description="Salutation sent event",
dataShape="SalutationSentEvent")
```

Subscriptions

You can add subscriptions to an entity in an extension to perform custom behavior when an event is fired. To add a subscription, use the `@ThingworxSubscriptions` annotation.

Tip

If you are using the Eclipse Plugin, the **Add Subscriptions** action will add the necessary annotations described below to the Java source file. For more information, see Adding Services, Properties, Configuration Tables, Subscriptions, and Events.

@ ThingworxSubscription Attribute	Description
source	The name of the entity, which is the source of the event. A blank value indicates that the event originates from the entity itself.
eventName	The name of the event to which you want to subscribe.
sourceProperty	The name of the property associated with the event. It's only applicable for the Alert, AlertAck, AlertReset, and DataChange events.
handler	The name of the service that should be invoked whenever the subscription receives an event. The service should have the following signature and be annotated appropriately as a ThingWorx service: `public void HandleEvent(InfoTable eventData, String eventName, DateTime eventTime, String source, String sourceProperty)`
enabled	Indicates if the subscription should listen for events

For more information, see the Javadoc documentation for `com.thingworx.metadata.annotations.ThingworxSubscription`.

For example:

```
@ThingworxSubscriptions( subscriptions = {
 @ThingworxSubscription( source = "", eventName = "MyEvent",
 sourceProperty = "", handler = "MyService", enabled = "true" )
})
```

Extension Migrators

When a new version of an extension contains model changes for one or more of its entities, the previous version of the extension and any entities created from that extension must be deleted before installing the new version of the extension. To avoid having to recreate the entities after the new extension is installed, you can include an extension migrator with the extension, which imports the entities created with the previous version.

To install a new version of an extension in this case, do the following:

1. Export all entities and data from the ThingWorx system.
2. Install the new version of the extension on a clean ThingWorx system.
3. Import the entities and data that were exported from the original system.
4. Optionally, restart ThingWorx or specific things or data tables. Restarting will trigger services to review, update, and clean up data that has been imported. It can also eliminate problems with loaded classes.

The extension migrator will run during the import process so that the entities from the previous version will be imported properly and match the model from the new extension.

To create a custom migrator, create a Java class that extends `com.thingworx.migration.ExtensionMigratorBase` and update the metadata.xml file by setting the `migratorClass` attribute on the `<ExtensionPackage>` element. For more information, see metadata.xml File.

Tip

If you are using the Eclipse Plugin, the **New Extension Migrator** action will generate the Java source file with the appropriate annotations and automatically update the metadata.xml file. For more information, see Adding an Extension Migrator.

The following method must be implemented in the migrator to complete the migration depending on the changes between the versions of the extension. See the Javadoc documentation for the `ExtensionMigratorBase` class for more information.

```
public void migrate(ImportedEntityCollection imports)
 throws Exception;
```

When importing the entities into the system containing the new version of the extension, the migrator will run.

Note

The extension migrator will only be invoked when importing all entities from the previous ThingWorx system. It will not be invoked when importing individual or subsets of entities.

Note

Extension migrators are designed for migrating entities only. Stream data associated with extension entities must be updated by other means before importing into the new system.

Third-Party JAR Files

The `/lib` folder contains the JAR file including the custom Java classes written for the extension and additional third-party JAR files required by the extension.

For example:

```
/lib
 /MyExtension.jar
 /ThirdPartyLib.jar
```

For each of these JAR files, you must create an entry in the metadata.xml file. For more information, see metadata.xml File.

Tip

If you are using the Eclipse Plugin, the **New Jar Resource** action will update the metadata.xml file automatically. For more information, see Adding Third-Party JAR Files.

Caution

Do not include third-party JAR files from your extension that are already included in the ThingWorx Platform.

Adding Custom Widgets

The `/ui` folder contains the files needed to define custom widgets that are used when building and running mashups. Each widget should be placed in its own subfolder of `/ui`. The following files are needed to define a widget:

- <widgetname>.ide.css

  Style sheet file that defines the look and feel of the widget in the Mashup Builder

- <widgetname>.ide.js

  JavaScript file that defines the widget and its behavior in the Mashup Builder

- <widgetname>.ide.png

  Icon used for the widget in the **Widgets** tab in the Mashup Builder, which should be 16 by 16 pixels

- <widgetname>.runtime.css

  Style sheet file that defines the look and feel of the widget when viewing the mashup

- <widgetname>.runtime.js

  JavaScript file that defines the widget and its behavior when viewing the mashup

Tip

If you are using the Eclipse Plugin, the **New Widget** action will generate the source files and automatically update the metadata.xml file. For more information, see Creating Widgets.

For an example of a widget and more information about the functions that a widget can implement and the APIs that it can call, see Widget Implementation Guide.

Third-Party JavaScript Libraries

If the custom widget needs to use third-party JavaScript libraries, the best practice is to create a subfolder in the widget folder (for example, `/ui/<widgetname>/<jslibrary>/`) and put the third-party library files there. These files can be referenced from the ide.js and runtime.js files using the following relative path:

```
../Common/extensions/<extensionname>/ui/<widgetname>/<jslibrary>/
```

Extension Import and Management

Importing an Extension into ThingWorx

Once an extension is packaged correctly, you can import it using ThingWorx Composer:

1. Go to **Import/Export** ▸ **Import**.

   The **Import** screen appears.

2. From the **Import Option** dropdown, select **Extension**.
3. Click **Browse** to select the .zip file for your extension.
4. If you want to check the extension before importing, click **Validate**.

   Validation results appear and any errors are described.

5. Click **Import**.

   If the extension can be imported without error, an **Import Successful** message appears. The ThingWorx Platform prompts you to refresh the window to load the new functionality.

Note

If you import an updated version of an extension that contains a JAR file, you will see a message that instructs you to restart the platform to complete the upgrade.

Note

If a menu is imported in an extension, the *Group Association* property on menu and the *Groups* property on menu items are editable. If you import an updated version of the extension that contains the menu, changes made in ThingWorx will be merged with any changes that were made in the extension.

Note

Configuration tables that are imported in an extension are editable. If you import an updated version of an extension that contains a configuration table, changes made in ThingWorx will be merged with any changes that were made in the extension.

Removing Old Versions and Uploading New Versions

The process of developing extensions involves multiple iterations of importing an extension, testing it, and making changes.

To remove an extension, do the following:

1. Go to **Import/Export** ▸ **Manage Extensions**.

   The **Manage Extensions** screen appears.
2. Select the extension in the list of installed extension packages and click **Delete**.

Note

Before you remove the extension, you must delete new entities that reference the entities provided by the extension. You can export these entities first so that you can import them after the new version of the extension is installed.

Java-based entities from the extension are loaded into memory by the Platform every time you upload an extension. Therefore, each class will remain there, even after you remove the extension.

Tip

To avoid clashing versions of the same class, restart Tomcat between version changes in order to remove the classes that were previously loaded in the Platform.

If there are bugs in the Java code for entities provided by the extension, ThingWorx may be unable to completely remove them. If this occurs, delete the `ThingworxStorage` directory between version changes since you may be unable to properly remove an extension. If there is anything on your development Platform instance that you want to save, we recommend that you Back Up Storage Before Delete before testing new code.

Entity Characteristic Support

The following table shows which entities support Services, Properties, Configuration Tables, Subscriptions, and Events:

Entity Type	Services	Properties	Configu-ration Tables	Subscrip-tions	Events
Thing	Supported	Supported	Supported by REST API	Supported	Supported
Thing Template	Supported	Supported	Supported by REST API	Supported	Supported
Mashup			Supported by REST API		
Thing Shape	Supported	Supported	Supported by REST API	Supported	Supported
Resource	Supported				
Extension Migrator					
Authentica-tor	Supported		Supported by REST API		
Directory Service			Supported by REST API		
Script Function Library					

4

Using the Eclipse Plugin

Installing the Eclipse Plugin for ThingWorx Extension Development

To install the plugin for ThingWorx Extension Development into your local instance of the Eclipse IDE for Java EE Developers, Mars 4.5 release or newer, do the following:

1. Download the `Eclipse Plugin for ThingWorx Extensions` zip file from the ThingWorx Marketplace.
2. Open Eclipse and choose a workspace.
3. Choose **Help ▸ Install New Software...**.

   The **Install** screen appears.
4. Click **Add...**.

   The **Add Repository** screen appears.
5. Choose the download location for the zip file (for example, `thingworx-eclipse-plugin-[version].zip`).
6. Select **ThingWorx Extension Builder**.

   You may have to deselect the **Group items by category** checkbox for the ThingWorx Extension Builder plugin to appear in the list.
7. Click **Next**.
8. Accept the license and finish the installation.
9. Click **OK** to acknowledge the Eclipse security warning.
10. Restart Eclipse.
11. To verify your install in Eclipse, choose **Help ▸ Installation Details**. **ThingWorx Extension Builder** appears in the list of installed software.

Creating an Extension Project

To get started using the Eclipse Plugin for ThingWorx Extension Development, do the following:

1. In Eclipse, choose **File ▸ New ▸ Project...**.
2. In the **New Project** screen, expand the **ThingWorx** menu, select **ThingWorx Extension Project**, and click **Next**.
3. Enter a project name and browse to and select the latest `ThingWorx-Extension-SDK-[version]-latest.zip` file.
4. Select **Gradle** or **Ant** as the build framework for the extension.

> **Note**
>
> To use Gradle to build the extension, the Gradle STS plugin must be installed in Eclipse. Only the Gradle STS plugin is currently supported.

A build file corresponding to the selected framework is created. For more information, see Building the Extension.

5. You can enter a vendor name, update the package version, and then click **Finish**.

   The project appears in the **Package Explorer**, and you are now working in the **ThingWorx Extension** perspective.

6. To view and edit the properties for your project, right-click your project folder in the **Package Explorer** and choose **Properties**.

   On the **Properties** screen under the **ThingWorx Extension** menu, you can enter a list of extensions on which your extension depends and their versions in the **Depends On Extensions** field. For example, `abc-extension:2.1.0,xyz-extension:1.5.4`.

Importing Existing Extensions

To import an existing extension into the plugin, do the following:

1. In Eclipse, make sure you are in the ThingWorx Extension perspective. To do so, choose **Window** ▸ **Perspective** ▸ **Open Perspective** ▸ **ThingWorx Extension**.
2. Choose **File** ▸ **Import**.
3. On the **Import** screen, choose **ThingWorx** ▸ **Extension Project** and click **Next**.
4. Browse to and select your extension zip file.
5. Select the latest `Thingworx-Extension-SDK-[version]-latest.zip` file and click **Finish**.

   Your project appears in the **Package Explorer**.

> **Note**
>
> If you want to update the existing source code, you must import the source files into the Eclipse project.

6. To view and edit the properties for your project, right-click your project folder in the **Package Explorer** and choose **Properties**.

Creating Entities

1. To create an entity, choose the **ThingWorx** menu and select the entity type you want to create.
2. Select or enter your source folder and package.
3. Enter a name.

> **Note**
>
> We recommend that you add a unique prefix to extension entity names to avoid conflicts with other extension entities, as in `<MyExtension>.<MyEntityName>`.

4. If you want to edit the aspects of your entity, click **Next** and edit the available aspects. Or, to use the defaults, click **Finish**.

   The aspects will be different for each entity type.

   The Java source file is created for you in the specified package and the metadata.xml file is updated automatically.

Adding Services, Properties, Configuration Tables, Subscriptions, and Events

To add services, properties, and other annotations to an entity, do the following:

1. In **Package Explorer**, right-click on the Java file and choose the **ThingWorx Source** menu.

   OR

   In the Java editor of the entity, right-click to access the **ThingWorx Source** menu.

   From the submenu, you can choose to add a service, property, configuration table, subscription, or event.
2. Enter the necessary information in the wizard and click **Finish**.

   The Java and metadata.xml files are updated with the necessary annotations and XML elements.

Creating Widgets

1. Choose the **ThingWorx** menu and select **New Widget**.
2. Select the parent project.
3. Enter a name and description.
4. Click **Finish**.

   A new folder under the `/ui` folder is created and contains the JavaScript and CSS files for the widget. The metadata.xml file is updated automatically.

The generated JavaScript files contain a minimal implementation of the functions needed to produce a working widget. For more information, see Adding Custom Widgets and Appendix B: Widget Implementation Guide.

Adding Third-Party JAR Files

1. Choose the **Widget** menu and select **New Jar Resource**.
2. Select the parent project.
3. Browse to and select the JAR file you want to add, and enter a description.
4. Click **Finish**.

   The JAR file is added to the `/lib` folder and the metadata.xml file is updated automatically.

Adding an Extension Migrator

1. To create a migrator class for the extension, choose the **ThingWorx** menu and select **New Extension Migrator**.
2. Select or enter your source folder and package.
3. Enter a name.
4. Click **Finish**.

   The Java source file is created and the metadata.xml file is updated automatically.

Note

If a migrator has already been created, the previous migrator class name will be replaced in the metadata.xml file with the new migrator class name. The previous migrator Java file will not be changed.

Importing Composer-Created Entities

1. Choose **File** ▸ **Import**.
2. On the **Import** screen, choose **ThingWorx** ▸ **Entities** and click **Next**.
3. Browse to the folder that contains the entity folders that were exported from Composer (for example, `ThingworxStorage/repository/<RepositoryName>/<EntitiesFolder>`).
4. If necessary, select or deselect individual XML files to be imported.
5. Select the project into which you want to import and click **Finish**.

   The selected XML files are imported under the `/Entities` folder in the extension project.

Tip

When exporting entities from Composer, you should tag all entities created for the extension and then export them using the **Export Source Control Entities** action. In the **Export Source Control Entities** window, specify the tag and a file repository and enter `/Entities` as the path. You can select the resulting `ThingworxStorage/repository/<RepositoryName>/Entities` folder in the **Import** wizard in Eclipse.

Building Extensions

Gradle Build

In the Package Explorer, right-click *build.gradle* and choose **Run As** ▸ **Gradle Build**.

If the **Edit Configuration** window appears, enter `clean build` in the task editor, click **Apply** and then **Run**.

After refreshing your project, a zip file appears under **build** ▸ **distributions**. This is a working extension that you can import into ThingWorx.

Ant Build

In the Package Explorer, right-click *build-extension.xml* and choose **Run As** ▸ **Ant Build**.

After refreshing your project, a zip file appears under **build** ▸ **distributions**. This is a working extension that you can import into ThingWorx.

Deleting Entities and Widgets

1. In the **Package Explorer**, select the source file to be deleted and press `Delete` or right-click and select **Delete** from the menu.

   The delete confirmation window appears.

2. To see what changes will occur, click **Preview**, or click **OK** to delete the file.

The file is deleted from the project and the metadata.xml file is updated.

Note

When you delete one of the widget files generated by the plugin, the widget folder and its contents are deleted. To keep a file that was manually added to the widget folder, use the **Preview** option to deselect the files and folders so they are not deleted.

Tip

The XML definitions in the metadata.xml file must be in sync with the contents of the extension. If there is an entry in metadata.xml file that is missing a corresponding entity source file, the extension will not import correctly. Therefore, do not deselect the **Remove XML definition from project's metadata.xml** option in the **Preview** window.

5

Best Practices

There may be many different ways to accomplish a goal in extensions. The following section outlines best practices for developing successful extensions.

Use the Eclipse Plugin

The Eclipse plugin makes it easier to develop and build extensions by providing actions to automatically generate source files, annotations, and methods and update the metadata file to ensure proper configuration of the extension. These features allow developers to focus on developing functionality in their extension, rather than worrying about the getting the syntax and format of annotations and the metadata file correct.

The Golden Rule: Use as Few External JARs as Possible

Though ThingWorx allows the users to include third-party libraries in their code, it is recommended that you avoid common JARs as much as you can. You should use the JARs that are packaged with the SDK (for example, the slf4j logger) when applicable. The nature of the Platform development process allows for multiple sources to attempt to use the same JAR files. When this happens, there will either be a conflict when uploading an extension to the Platform, or functionality will be different because the wrong version is used. This leads to future versions of the Platform requiring updates to extensions to allow them to work. Similarly, it may mean that a customer cannot use your extension with someone else's extension at the same time.

Develop on a Clean Platform Instance

ThingWorx does not provide safeguards or sand-boxing of your code's execution. It is possible to put your environment in an unstable state. You may see some errors in the development process which will require you to restart Tomcat (always try this first). Occasionally you may even be forced to remove the `ThingworxStorage` directory completely. For this reason it is recommended that you develop on a clean instance so that no work is lost by removing the `ThingWorxStorage` directory.

Choosing Between a Thing Template and Thing Shape

When possible and sensible, use a thing shape to encapsulate specific functionality. By using thing shapes, you have the ability to easily add the new functionality to existing things in ThingWorx.

For example, if you have already implemented your assets using custom thing templates and things, it is fairly simple to add a thing shape from a new extension to those thing templates. However, if the properties or services were defined

directly on a thing template, you would need to recreate all of your assets to use the new functionality, because the base thing template cannot be changed for a thing template or a thing.

If you want to create subscriptions, you might choose to create thing templates that implement one or more thing shapes for convenience, as subscriptions cannot be created on thing shapes.

For more information about these concepts, see the *ThingWorx Model Definition and Composer* section in the ThingWorx Help Center.

Customizing Media Entities

Media entities are images and icons used in menus, style definitions, and mashups. For more information, see Media Entities in the ThingWorx Platform Help Center.

You can provide branded content by changing default logos (images). You can configure your media entities to be bound to the Image widget via a service. That service can provide a default media entity or a customer-provided media entity.

Customizing Mashups

Mashups can be bound to a Contained Mashup widget via a service. You can copy a default (non-editable) mashup, modify it, and set the configuration to the copy.

Caution

Mashups are complex, and customizing them in this way can introduce problems for extension upgrades.

Customizing Style Definitions

Since it is currently not possible to bind style definitions to a widget via a service, you could make style definitions editable in order to customize, or rebrand, them. You should limit the number of editable style definitions.

Note

You should bundle editable style definitions (and all editable entities) in a separate extension.

Make Your Entities Non-Editable

When creating entities, such as mashups, style definitions, and so on, it is a best practice to make them non-editable. Only non-editable entities can be upgraded in place; editable entities cannot. New entities are non-editable by default.

If you need to make it possible to customize the extension, such as by adding a custom logo, consider whether an alternative approach would work:

- Can thing configuration be used instead? A non-editable thing can still have configuration table changes.
- Can a service be used to lookup an configuration to provide a media entity or embedded mashup?

If you must use editable entities, following are best practices:

- Minimize editable locations on a mashup by using embedded mashups.
- When planning an upgrade, perform a test migration to determine if any information might be lost during that process, such as tags.

To make an entity editable, set `aspect.isEditableExtensionObject= "true"`.

Tag Your Extension Entities

Tag your extension entities during development, using a new vocabulary of tags specifically for your extension. Clear and consistent tags are helpful when you want to export entities for an extension, modify visibility, or identify entities to upgrade.

Add Localization Tokens

If your extension will be localized, you should plan for this when developing your user interface. When you need to present non-data text to your users, use a localization token and create new tokens as needed. It is much easier to do this work during initial development than having to find and replace labels in your mashups later.

If your extension has a configuration option with a data shape that includes friendly names in its aspects, you must provide localization tokens to localize those prompts in ThingWorx Composer. If the name includes a period (.), it will be converted to an underscore. For example, if `aspects.friendlyName = "myNamespace.myKey"`, the localization token lookup would be `myNamespace_myKey`.

To avoid conflicts with system tokens or tokens from other extensions, consider using a prefix or suffix that is specific to your extension. For example, instead of `NoNameProvided`, you might use `MyExtension.NoNameProvided`.

For more information, see the *Localization Tables* topic in the ThingWorx Help Center.

These tokens must be included in your extension. The easiest way to do this is to use the Eclipse plugin to import the localization table XML entity file. You can also add these tokens manually by following the steps in the *Packaging Extensions with Localization Tables* topic in the ThingWorx Help Center.

If a specified token is not found in the localization table, `tw.friendly-names.[`*`your value`*`]` is displayed in ThingWorx Composer.

Avoid Tight-Coupling to Other Extensions

Avoid tight-coupling your extension to other extensions. If your extension is tight-coupled to another extension that needs to be upgraded to a new major version, you will be forced to delete your extension and its entire chain of dependencies before you can perform the upgrade.

The best way to do avoid tight-coupling is to create the things and thing templates that you are need so that they are not considered part of your extension. This approach is recommended instead of using out-of-the-box things or thing templates that derive from other extension thing templates or implement other extension thing shapes

Note

It might be difficult to avoid tight-coupling when using widgets from other extensions in your extension mashups.

Example

Suppose that you created a thing that extends from the MailServer thing template in Mail_Extensions. When you import a new major version of Mail_Extensions, the platform forces you to delete the Mail_Extensions, which also forces you to delete ThingWorx Utilities (and down the dependency chain) before you can install the upgrade.

Instead, it would be better create you own thing that derives from MailServer and configure it in ThingWorx Utililities. Then, if you import a new major version of Mail_Extensions, you would only need to delete the thing that you created, and not ThingWorx Utilities or other dependencies.

Plan for Entity Visibility

If you have multiple organizations, make a plan for extension entity visibility, and ensure that you know how your user groups and organizations are structured. For example, it is important to know if the default Users group is present in the default Everyone organization.

If you have extension user groups, make sure they have a matching organization. Add that organization to the visibility of entities, such as mashups, that the user group needs to access.

For more information, see the *Visibility in Organizations* topic in the ThingWorx Help Center.

Example: User Group to Organization

In ThingWorx Utilities, there is a TW.RSM.RemoteServices user group that is in a TW.RSM.RemoteServices organization. The user group has runtime permissions to several services used in Software Content Management and Remote Access and Control, and the organization gives those same users visibility to the related mashups, data shapes, and things.

Even if your extension is not meant for a particular user group, you should plan to accommodate non-administrator users.

ThingWorx Utilities Visibility

ThingWorx Utilities includes a service that helps you assign visibility to an organization or organizational unit. This service finds extension entities and assigns visibility to that organization. For more information, see the "Visibility Support for ThingWorx Utilities" section of the *ThingWorx Utilities Installation Guide*.

Back Up Storage Before Delete

To back up your current `ThingworxStorage` directory, rename the directory. This will save the current status and a new `ThingworxStorage` directory will be generated once the Platform is restarted. You can then restore your old state by renaming the directory to "`ThingworxStorage`".

Note

If you do not back up your storage, make sure that any entities you create on Composer are exported into xml format because they will be deleted with the storage directory.

Logging

When you are trying to figure out what is happening on the Platform, use logging. Use ThingWorx `LogUtilities` to get a specific logger. There are multiple kinds of loggers and log levels used in the ThingWorx Platform, but we recommend that you use the application or script loggers for logging anything from inside extension services.

For example:

```
protected static Logger logger =
 LogUtilities.getInstance().getApplicationLogger(myThing.class);
logger.info("This is an information message");
logger.error("This is an error message");
```

6

Troubleshooting and Debugging

ThingTemplate Does Not Exist After Successful Import

This can happen when there is an issue creating the ThingTemplate once it has been imported, but it is not a fatal exception. The most common cause of this is a missing JAR that is required for the ThingTemplate class. If this happens, check your `metadata.xml` file for the JAR declaration of a required JAR. Then make sure the JAR is present in the extensions `lib/common` directory.

JAR Conflict on Import

This can occur when a JAR you are attempting to use is already loaded on ThingWorx Platform. Usually you can remove the JAR from your `metadata.xml` file to fix the problem. However, this can be risky because different versions of the same JAR can cause conflicts in functionality. This could affect your extension or the Platform itself. The best solution is to try to avoid using this JAR, if possible.

Updated Extension Retains Old Version's Functionality

The cause of this problem is the loaded classes from the previous version. This can be fixed by restarting Tomcat.

Failed to Create Data Shape

This will happen if there is an incorrect declaration in a Java annotation. For example, you define a property as `BOOL` or `Boolean` instead of `BOOLEAN`. Make sure you are using the proper string constants in a declaration and note that all base type values are case-sensitive.

Thing Does Not Run After Create or Save

This usually means that a thing was created or saved based on a thing template that has an error in its initialization method. Make sure you are catching exceptions and logging them so you can deduce the reason for the initialization failure.

Debugging an Issue When Importing

When importing an extension, classes are loaded from provided JARs, entities are created, and multiple background processes execute at the same time. Sometimes this can cause unexpected errors on import. Whenever an import fails, or succeeds but is missing something, it is usually best to check the Platform's Application Log. In the vast majority of cases, you will find an error explaining what went wrong with your import. To get to this, and other logs, use the **Monitor** button at the top right of Composer.

Connecting a Debug Port to Tomcat

The most useful way of debugging involves connecting a debug port to your Tomcat instance. This allows you to connect to the Platform from inside an IDE and add breakpoints to the code you have uploaded. This will let you trigger a service, set a property, or save a thing and track what happens inside your code as it executes. It is important to note that the code you uploaded must be exactly the same as the code in which you have breakpoints. If it is not, then the breakpoints you add will be associated with different lines of code and may have different values and functionality than you would expect.

To add a debug port to Tomcat, you must add a Java option on startup. There are multiple ways to do this and it will depend on how you are launching Tomcat. Since there are many ways to properly configure this, it is recommended that you research your particular Tomcat and IDE setup.

See Example Using Tomcat Monitor GUI and Eclipse.

7

Appendix A: Examples

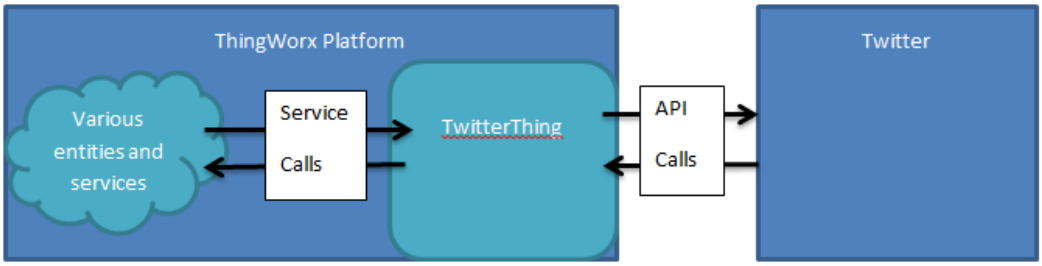
Platform Integration Example using Twitter

A common reason for creating an extension is to create a connection between ThingWorx and external platforms through APIs and database connectors. This is commonly done by creating a connector between ThingWorx and a web resource. In the following example we will show how this is done in the Twitter Extension.

The bulk of the work done by the Twitter extension is through the TwitterThing thing template, which is used as a connector. In TwitterThing, there are various configuration values and services that allow the ThingWorx Platform to connect to Twitter.

The configuration table includes four values that need to be set in order for the extension to work: consumerSecret, consumerKey, accessToken, and accessTokenSecret. The configuration values are consumed by the service calls to connect to the proper host and add authentication.

By using these configuration values and a predefined API, the Twitter extension exposes ThingWorx Services that interface with Twitter directly. This architecture results in a single template, or a set of Things extending this template, that expose API functionality to entities and services within ThingWorx Platform.

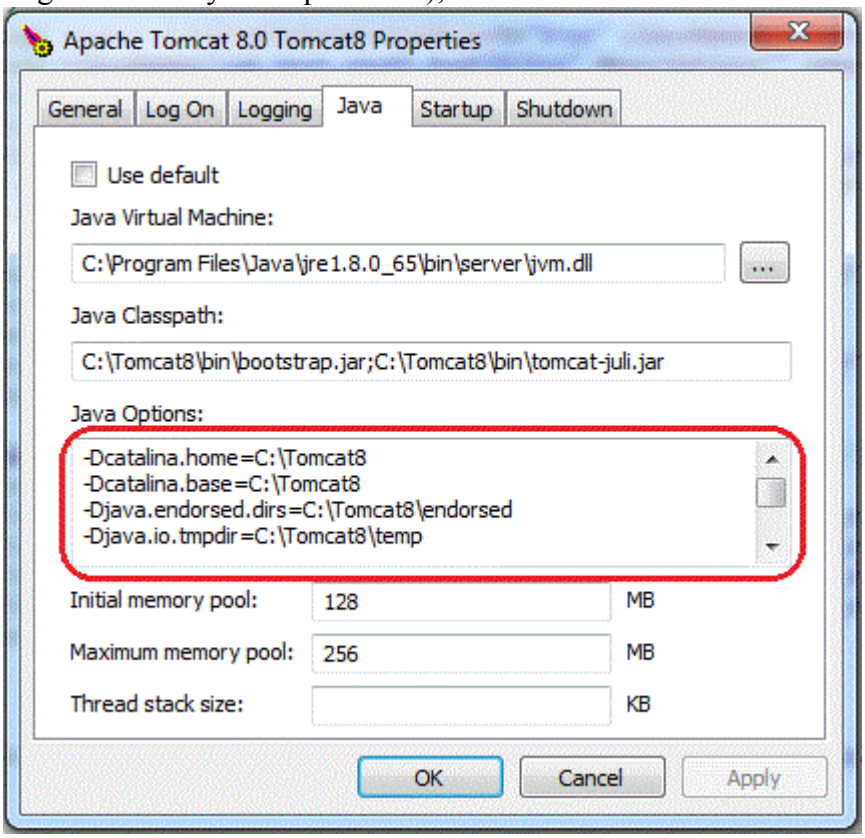


Example using Tomcat Monitor GUI and Eclipse

1. Set up Tomcat to start in debug mode and open a port. When using the Tomcat Monitor GUI, this can be done in the **Java Options** section of the **Java** tab when editing Properties (shown below). The line you want to enter in Java Options is as follows:

   ```
   -agentlib:jdwp=transport=dt_socket,address=1043,server=y,suspend=n
   ```

Enter any available port value for `address` (do not use well-known ports that might be used by other processes), and note this value for the next step.



2. Set up a remote debug connection in your IDE. This should be done differently depending on the IDE you are using.

   If you are using Eclipse:

   a. Go to **Run ▸ Debug Configurations**.
   b. Select **Remote Java Applications** and create a new configuration (**right click ▸ New**).
   c. In the **Project** field, browse for and select your project.
   d. In the **Host** field, enter the host of your Tomcat instance.
   e. In the **Port** field, enter the debug port to which you intend to connect (this should match the port/address you entered in step 1).
   f. Run debug after Tomcat has started in debug mode, and you will be connected.

HelloWorldThing

```
package com.helloinc.things.helloworld;
```

```
import org.slf4j.Logger;

import com.thingworx.data.util.InfoTableInstanceFactory;
import com.thingworx.logging.LogUtilities;
import com.thingworx.metadata.annotations.ThingworxEventDefinition;
import com.thingworx.metadata.annotations.ThingworxEventDefinitions;
import com.thingworx.metadata.annotations.ThingworxPropertyDefinition;
import com.thingworx.metadata.annotations.ThingworxPropertyDefinitions;
import com.thingworx.metadata.annotations.ThingworxServiceDefinition;
import com.thingworx.metadata.annotations.ThingworxServiceParameter;
import com.thingworx.metadata.annotations.ThingworxServiceResult;
import com.thingworx.things.Thing;
import com.thingworx.things.events.ThingworxEvent;
import com.thingworx.types.InfoTable;
import com.thingworx.types.collections.ValueCollection;
import com.thingworx.types.primitives.NumberPrimitive;
import com.thingworx.types.primitives.StringPrimitive;
import com.thingworx.webservices.context.ThreadLocalContext;

@ThingworxPropertyDefinitions(
  properties = {
    @ThingworxPropertyDefinition(name = "StringProperty1",
       description = "Sample string property",
       baseType = "STRING",
       aspects = { "isPersistent:false", "isReadOnly:false" }),
    @ThingworxPropertyDefinition(name = "NumberProperty1",
       description = "Sample number property",
       baseType = "NUMBER",
       aspects = { "isPersistent:false", "isReadOnly:false" }) })

@ThingworxEventDefinitions(
  events = {
    @ThingworxEventDefinition(name = "SalutationSent",
       description = "Salutation sent event",
       dataShape = "SalutationSentEvent") })

public class HelloWorldThing extends Thing {

  protected static Logger _logger =
LogUtilities.getInstance().getApplicationLogger(HelloWorldThing.class);

  // local property values
  private String _stringProperty1 = "";
  private Double _numberProperty1 = 0.0;

  protected void initializeThing() throws Exception {

      _stringProperty1 = ((StringPrimitive)
this.getPropertyValue("StringProperty1")).getValue();
   _numberProperty1 = ((NumberPrimitive)
this.getPropertyValue("NumberProperty1")).getValue();
```

```
    }

    @ThingworxServiceDefinition(name = "SayHello", description = "Hello world")
    @ThingworxServiceResult(name = "Result",
      description = "Result", baseType = "STRING")
    public String SayHello() throws Exception {

      return "Hello world.";
    }

    @ThingworxServiceDefinition(name = "Greeting", description = "Hello world")
    @ThingworxServiceResult(name = "Result",
       description = "Result", baseType = "STRING")
    public String Greeting(@ThingworxServiceParameter(name = "userToGreet",
description = "The user you're greeting",
                    baseType = "USERNAME") String userToGreet)throws Exception {

        fireSalutationSentEvent(userToGreet);
        return "Hello " + userToGreet + ". How are you?";
    }

    // Event Firing Example
    private void fireSalutationSentEvent(String userName) throws Exception {
        ThingworxEvent event = new ThingworxEvent();
        event.setTraceActive(ThreadLocalContext.isTraceActive());
        event.setSecurityContext(ThreadLocalContext.getSecurityContext());
        event.setSource(getName());
        event.setEventName("SalutationSent");

        // the name parameter isn't really used
        InfoTable data =
InfoTableInstanceFactory.createInfoTableFromDataShape("SalutationSentEvent");

        ValueCollection values = new ValueCollection();
        values.put("userGreeted", new StringPrimitive(userName));

        data.addRow(values);

        event.setEventData(data);

        this.dispatchBackgroundEvent(event);
    }

}
```

GoodByeThing

```
package com.helloinc.things.goodbye;
```

```
import org.slf4j.Logger;

import com.thingworx.entities.utils.EntityUtilities;
import com.thingworx.entities.utils.ThingUtilities;
import com.thingworx.logging.LogUtilities;
import com.thingworx.metadata.annotations.ThingworxConfigurationTableDefinition;
import com.thingworx.metadata.annotations.ThingworxConfigurationTableDefinitions;
import com.thingworx.metadata.annotations.ThingworxDataShapeDefinition;
import com.thingworx.metadata.annotations.ThingworxFieldDefinition;
import com.thingworx.metadata.annotations.ThingworxServiceDefinition;
import com.thingworx.metadata.annotations.ThingworxServiceParameter;
import com.thingworx.metadata.annotations.ThingworxServiceResult;
import com.thingworx.relationships.RelationshipTypes.ThingworxRelationshipTypes;
import com.thingworx.resources.entities.EntityServices;
import com.thingworx.things.Thing;
import com.thingworx.types.ConfigurationTable;
import com.thingworx.types.InfoTable;
import com.thingworx.types.collections.ValueCollection;
import com.thingworx.types.primitives.NumberPrimitive;


@ThingworxConfigurationTableDefinitions(tables = {
@ThingworxConfigurationTableDefinition(name="ConfigTableExample1",
    description="Example 1 config table", isMultiRow=false,
    dataShape = @ThingworxDataShapeDefinition(
        fields = {
          @ThingworxFieldDefinition(name="field1",description="",baseType="STRING"),
          @ThingworxFieldDefinition(name="field2",description="",baseType="NUMBER"),
          @ThingworxFieldDefinition(name="field3",description="",baseType="BOOLEAN"),
          @ThingworxFieldDefinition(name="field4",description="",baseType="USERNAME")})),
@ThingworxConfigurationTableDefinition(name="ConfigTableExample2",
description="Example 2 config table", isMultiRow=true,
    dataShape = @ThingworxDataShapeDefinition(
        fields = {
          @ThingworxFieldDefinition(name="columnA",description="",baseType="STRING"),
          @ThingworxFieldDefinition(name="columnB",description="",baseType="NUMBER"),
          @ThingworxFieldDefinition(name="columnC",description="",baseType="BOOLEAN"),
          @ThingworxFieldDefinition(name="columnD",description="",baseType="USERNAME") })) })

public class GoodByeThing extends Thing {

    protected static Logger _logger =
 LogUtilities.getInstance().getApplicationLogger(GoodByeThing.class);

    @Override
```

```
   public void initializeThing() throws Exception {
       super.initializeThing();

       _logger.warn("****** Initializing GoodByeThingInstance " + getName());

       // Examples of getting the configuration table values
       String field1Value = (String)getConfigurationSetting("ConfigTableExample1", "field1");
       Double field2Value = (Double)getConfigurationSetting("ConfigTableExample1", "field2");
       Boolean field3Value = (Boolean)getConfigurationSetting("ConfigTableExample1", "field3");
       String field4Value = (String)getConfigurationSetting("ConfigTableExample1", "field4");

       // Example of setting a configuration table value
       setConfigurationSetting("ConfigTableExample1", "field2", (field2Value + 2.1));
       SaveConfigurationTables();
       // Another example
       setConfigurationSetting("ConfigTableExample1", "field2",
((Double)getConfigurationSetting("ConfigTableExample1", "field2") + 5.5));
        SaveConfigurationTables();

       // Example of getting multi-row configuration table data
       ConfigurationTable tableTwo = getConfigurationTable("ConfigTableExample2");

       _logger.warn("****** Iterating " + tableTwo.getName() + " of goodbye thing instance " +
getName());
       _logger.warn("****** " + tableTwo.getName() + " has " + tableTwo.getLength() + " rows");

       int counter = 0;
       for(ValueCollection row : tableTwo.getRows()) {

           String columnA = row.getStringValue("columnA");
           Double columnB = (Double)row.getValue("columnB");
           Boolean columnC = (Boolean)row.getValue("columnC");
           String columnD = row.getStringValue("columnD");

           _logger.warn("columnA row " + counter++ + " value is " + columnA);
           _logger.warn("columnB row " + counter++ + " value is " + columnB);
           _logger.warn("columnC row " + counter++ + " value is " + columnC);
           _logger.warn("columnD row " + counter++ + " value is " + columnD);
       }
}

   // Property set of another Thing
       @ThingworxServiceDefinition( name="SetHelloWorldProperty",
description="Set a property on Hello world" )
       public void SetHelloWorldProperty(
         @ThingworxServiceParameter( name="thingToSet", description="The thing you're setting",
```

```
baseType="THINGNAME", aspects={"thingTemplate:HelloWorld"} ) String thingToSet,
         @ThingworxServiceParameter( name="numberPropertyToSet", description=
"The property name you're setting", baseType="STRING" ) String numberPropertyToSet,
         @ThingworxServiceParameter( name="numberValueToSet", description=
"The propert value you're setting", baseType="NUMBER") Double numberValueToSet) throws Exception {

         Thing helloThing = ThingUtilities.findThing(thingToSet);
         helloThing.setPropertyValue(numberPropertyToSet, new NumberPrimitive(numberValueToSet));
     }

        // Access a Resource
        @ThingworxServiceDefinition( name="AccessAResource",
description="Execute a service of a resource" )
        @ThingworxServiceResult( name="result", description=
"Matching entries", baseType="INFOTABLE", aspects={"dataShape:RootEntityList"} )
        public InfoTable AccessAResource() throws Exception {

       // Easiest way to do this as well as access any service of a Thing
       // is simply to call what you see in Composer.
       EntityServices entityService = (EntityServices)
EntityUtilities.findEntity("EntityServices",
ThingworxRelationshipTypes.Resource);

      return entityService.GetEntityList("Thing", "", null, null);
  }

}
```

Common Code Snippets

Getting Property and Configuration Values

Using local variables to keep track of properties and configuration values can be convenient, but it can also lead to incorrect values being used in various situations. This can be a tough problem to track down. Therefore, it is recommended that when you reference a thing's property or configuration value, you get the current value each time. This avoids problems with external sources setting values, but not having those values update in your Java code. Here are some examples of the correct ways to access a thing's property and configuration values from within the thing's class.

To access a thing's property:

```
StringPrimitive stringProperty1 =
 (StringPrimitive)this.getPropertyValue("StringProperty1");
```

or

```
String stringProperty1 =
 ((StringPrimitive)this.getPropertyValue("StringProperty1")).getValue();
```

To access a thing's configuration:

```
String field1Value =
 getStringConfigurationSetting("ConfigTableExample1", "field1");
```

Setting Property and Configuration Values

Setting property and configuration values is similar to getting properties. The biggest difference between the two is that configuration values need to be saved once they are set.

Example of setting a property value:

```
this.setPropertyValue(numberPropertyToSet,
 new NumberPrimitive(numberValueToSet));
```

Example of setting a configuration table value:

```
this.setConfigurationSetting("ConfigTableExample1", "field2", (field2Value + 2.1));
SaveConfigurationTables();
```

Another example of setting a configuration table value:

```
this.setConfigurationSetting("ConfigTableExample1", "field2",
 ((Double)getConfigurationSetting("ConfigTableExample1", "field2")
 + 5.5));
SaveConfigurationTables();
```

File Management

The file system for ThingWorx does not have a guaranteed static path that can be referenced. For this purpose, there is a FileRepositoryThing template that is used to store and access files. The File Repository thing has services, such as openFileForWrite() and openFileForRead(), that allow you to access files that are stored in unique locations based on the file repository thing you are using.

Similarly, if you need to upload a file with an extension, then it should be done through a JAR on the classpath, since the location of an uploaded file may not be known. Here is an example using the file repository thing and a file pulled from the classpath:

```
public String exportResource(String resourceName,
 FileRepositoryThing repo) throws Exception {

 InputStream stream = null;
 OutputStream resStreamOut = null;
 String jarFolder;

 try {
```

```
  stream = this.getClass().getClassLoader().
   getResourceAsStream(resourceName);
  if(stream == null) {
   throw new Exception("Cannot get resource \"" +
    resourceName + "\" from Jar file.");
  }

  int readBytes;
  byte[] buffer = new byte[4096];
  resStreamOut = repo.openFileForWrite(resourceName,
   FileRepositoryThing.FileMode.WRITE);
  while ((readBytes = stream.read(buffer)) > 0) {
   resStreamOut.write(buffer, 0, readBytes);
  }
 } catch (Exception ex) {
    throw ex;
 } finally {
     if(stream != null) {
      stream.close();
     }
     if(resStreamOut != null) {
      resStreamOut.close();
  }
 }
 return resourceName;
}
```

Sending HTTP Messages

While there are many convenient HTTP builders and clients to use, it is recommended that you avoid using them for the same reason we have our Golden Rule. Fortunately, this functionality is common and is already built into the Platform through the form of the ContentLoader. The Content loader takes many parameters in order to send an HTTP message, but many are often set to null because they are not needed. Here is a simple example of using the content loader to send an HTTP Post and get the response. In this case we are constructing a SOAP request.

```
public String sendSoapMessage(String host, String path, String soapAction,
 String content) throws Exception{

 ContentLoader loader = new ContentLoader();
 JSONObject headers = new JSONObject();
 headers.put("SOAPAction",soapAction);


 String result = "undefined";
```

```
 try{
   String url = "https://" + host + path;
   result = loader.PostText(url,content,"text/xml",null,
    null,headers,false,false,30000.0,null,null,null,
    false,null,null,null);
   String fault = checkForFaultCode(result);
   if (fault != "") {
    return fault;
   }
 } catch(SSLHandshakeException e){
  logger.error("Unexpected Error while sending request,
   possible incorrect hostname?: " + e.getMessage());
   return "Unexpected exception while sending message.
    Possible valid but incorrect host?";
 } catch(Exception e){
  logger.error("Unexpected Error while sending request: " +
   e.getMessage());
  throw e;
 }

 return result;
}
```

8

Appendix B: Widget Implementation Guide

This appendix details how the ThingWorx Mashup Builder and runtime interact with widgets, the functions that widgets can implement, and the APIs that they can call.

Referencing Third-Party JavaScript Libraries and Files

Third-party libraries, images, and other web artifacts needed by the widget should be placed in the /ui/<widgetname> folder or subfolders of that location. The *.ide.js and *.runtime.js files can then reference any of those artifacts via that relative path of:

../Common/extensions/<extensionname>/ui/<widgetname>/

For example, to include a third-party JavaScript library and CSS into your widget code, one can do the following:

```
if (!jQuery().qtip) {
  $("body").append('<script type="text/javascript"
   src="../Common/extensions/MyExtension/ui/mywidget/
   include/qtip/jquery.qtip.js"></script>');
  $("head").append('<link type="text/css"
   rel="stylesheet" href="
   ../Common/extensions/MyExtension/ui/mywidget/include/
   qtip/jquery.qtip.css" />');
}
```

Widget Example

Mashup Builder Code

The `<widgetname>.ide.js` file must implement several functions to work correctly in the Mashup Builder (see Widget API: Mashup Builder). Widgets can declare widget properties, services, and events in functions.

Below is sample code for a widget named *MyWidget* with a bindable string property named *DisplayText*.

```
TW.IDE.Widgets.mywidget = function () {
 this.widgetIconUrl = function() {
  return "../Common/extensions/MyExtension/ui/" +
   "mywidget/mywidget.ide.png";
 };

 this.widgetProperties = function () {
  return {
   name : "My Widget",
   description : "An example widget.",
   category : ["Common"],
   properties : {
    DisplayText: {
     baseType: "STRING",
     defaultValue: "Hello, World!",
     isBindingTarget: true
    }
```

```
   }
  }
 };

 this.renderHtml = function () {
  var mytext = this.getProperty('MyWidget Property');
  var config = {
   text: mytext
  }

  var widgetTemplate = _.template(
   '<div class="widget-content widget-mywidget">' +
    '<span class="DisplayText"><%- text %></span>' +
   '</div>'
  );
 return widgetTemplate(config);
 };
 this.afterSetProperty = function (name, value) {
  return true;
 };

};
```

Runtime Code

To handle the widget at runtime, you need methods to do the following:

- Render the HTML at runtime
- Set up bindings after rendering the HTML
- Handle property updates

Below is sample code of what the `<widgetname>.runtime.js` may look like.

```
TW.Runtime.Widgets.mywidget = function () {
 var valueElem;
 this.renderHtml = function () {
  var mytext = this.getProperty('MyWidget Property');
  var config = {
   text: mytext
  }
  var widgetTemplate = _.template(
   '<div class="widget-content widget-mywidget">' +
    '<span class="DisplayText"><%- text %></span>' +
   '</div>'
  );
  return widgetTemplate(config);
 };
 this.afterRender = function () {
  valueElem = this.jqElement.find(".DisplayText");
  valueElem.text(this.getProperty("DisplayText"));
```

```
 };
 this.updateProperty = function (updatePropertyInfo) {
  if (updatePropertyInfo.TargetProperty === "DisplayText") {
   valueElem.text(updatePropertyInfo.SinglePropertyValue);
   this.setProperty("DisplayText",
    updatePropertyInfo.SinglePropertyValue);
  }
 };
};
```

Additional Features

You can incorporate the following features into your widgets:

- Services that can be bound to events (such as *Click of a Button*, *Selected Rows Changed*, or *Service Completed*)
- Events that can be bound to various services (for example, invoke a service and navigate to a mashup)
- Properties can be bound out

You can access the full power of JavaScript and HTML in your widget code at runtime. Anything that can be done using HTML and JavaScript is available within your widget.

Widget API: Mashup Builder

Widget Lifecycle in the Mashup Builder

A widget has the following lifecycle within the Mashup Builder. During each lifecycle state, the specified functions on the widget are called by the Mashup Builder.

- Discovered

  The widget is being loaded into index.html and added to the widget toolbar/ palette.

  - widgetProperties()

    Called to get information about each widget (such as display name and description)

  - widgetEvents()

    Called to get information about the events each widget exposes

  - widgetServices()

    Called to get information about the services each widget exposes

- Created

The widget is dragged onto a mashup panel.

- afterload()

  Called after your object is loaded and properties have been restored from the file but before your object has been rendered

- Appended

  The widget is appended to the workspace DOM element.

  - renderHtml()

    Called to get an HTML fragment that will be inserted into the mashup DOM element

  - afterRender()

    Called after the HTML fragment representing the widget has been inserted into the mashup DOM element and a usable element ID has been assigned to the DOM element holding the widget content. The DOM element is then ready to be manipulated.

- Updated

  The widget is resized or updated in the widget property window.

  - beforeSetProperty()

    Called before any property is updated

  - afterSetProperty()

    Called after any property is updated

- Destroyed

  The widget is deleted from the mashup.

  - beforeDestroy()

    Called right before the widget's DOM element is removed and the widget is detached from its parent widget and deallocated. You should clean up resources (such as plugins and event handlers) acquired during the lifetime of the widget.

Mashup Builder APIs Available to the Widget

The following APIs can be accessed by a widget in the context of the Mashup Builder:

- this.jqElementId

  This is the DOM element ID of your object after renderHtml().

- this.jqElement

This is the jquery element.

- this.getProperty(name)
- this.setProperty(name,value)

  Note that every call to this function will call afterSetProperty() if it's defined in the widget.
- this.updatedProperties()

  This function should be called anytime properties are changed in the widget so that the Mashup Builder can update the widget properties window, the connections window, and so on.
- this.getInfotableMetadataForProperty(propertyName)

  If you need the infotable metadata for a property that you bound, you can get it by calling this API; it returns undefined if it is not bound.
- this.resetPropertyToDefaultValue(propertyName)

  This call resets the named property to its default value.
- this.removeBindingsFromPropertyAsTarget(propertyName)

  This call removes target data bindings from the propertyName. Use it only when the user has initiated an action that invalidates the property.
- this.removeBindingsFromPropertyAsSource(propertyName)

  This call removes source data bindings from the propertyName. Use this only when the user has initiated an action that invalidates the property.
- this.isPropertyBoundAsTarget(propertyName)

  This call returns a result that indicates if the property has been bound as a target. You can use it to determine if a property has been set or bound. For example, the blog widgets validate() function is:

```
this.validate = function () {
 var result = [];
 var blogNameConfigured = this.getProperty('Blog');

 if (blogNameConfigured === '' ||
  blogNameConfigured === undefined) {

  if (!this.isPropertyBoundAsTarget('Blog')) {
   result.push({ severity: 'warning',
    message: 'Blog is not bound for {target-id}' });
   }
  }
  return result;
```

```
}
```

- this.isPropertyBoundAsSource(propertyName)

  This call returns a result that indicates if the property has been bound as a source. You can use it to determine if a property has been bound to a target. For example, the checkbox widget's validate() function:

```
this.validate = function () {
 var result = [];

 if (!this.isPropertyBoundAsSource('State') &&
     !this.isPropertyBoundAsTarget('State')) {

  result.push({ severity: 'warning',
   message: 'State for {target-id} is not bound' });
 }

 return result;
}
```

Callbacks from Mashup Builder to Your Widget

The following functions on the widget are called by the Mashup Builder to control the widget's behavior.

- widgetProperties() - [required]

  Returns a JSON structure that defines the properties of the widget.

  Required properties are:

  - name

    The user-friendly widget name, as shown in the widget toolbar

  Optional properties are:

  - description - A description of the widget, which is used for its tooltip.
  - iconImage - File name of the widget icon/image
  - category - An array of strings for specifying one or more categories to which the widget belongs (such as "Common", "Charts", "Data", "Containers", and "Components"). This enables the user to filter widgets by type/category.
  - isResizable - true (default) or false
  - defaultBindingTargetProperty - Name of the property to use as the data/event binding target
  - borderWidth - If your widget has a border, set this property to the width of the border. This property helps to ensure pixel-perfect WYSIWG between

builder and runtime.If you set a border of one pixel on the widget-content element at design time, you are making the widget two pixels taller and two pixels wider (one pixel on each side). To account for this discrepancy, set the borderWidth property to make the design-time widget the same number of pixels smaller. This property places the border inside the widget that you created and makes the width and height in the widget properties accurate.

- isContainer - true or false (default). Controls whether an instance of this widget can be a container for other widget instances.
- customEditor – the name of the custom editor dialog to use for entering and editing the widget configuration. The system assumes there is a dialog named you created TW.IDE.Dialogs.<name>.
- customEditorMenuText - The text that appears on the flyout menu of the widget and the hover over text for the Configure Widget Properties button. For example, “Configure Grid Columns.”
- allowPositioning - true (default) or false
- supportsLabel - true or false (default). If true, the widget exposes a Label property whose value is used to create a text label that appears next to the widget in the Composer and at runtime.
- supportsAutoResize - true or false (default). If true, the widget can be placed in responsive containers (such as columns, rows, responsive tabs, responsive mashups).
- properties - A collection of JSON objects for the widget that describe the properties of the widget that can be modified when the widget is added to a mashup. These properties are displayed in the properties window of the Mashup Builder, with the name of each object used as the property name, and the corresponding attributes controlling how the property value is set.

  For example:

```
properties: {
 Prompt: {
   defaultValue: 'Search for...',
   baseType: STRING,
   isLocalizable: true
 },
 Width: {
   defaultValue: 120
 },
 Height: {
   defaultValue: 20,
   isEditable: false
 },
}
```

The following attributes can be specified for each property object:

- description - A description of the widget, which is used for its tooltip.
- baseType - The system base type name. If the baseType value is ‘FIELDNAME’, the widget property window displays a dropdown list of fields available in the INFOTABLE bound to the sourcePropertyName value based on the baseTypeRestriction. Other special baseTypes:
  - STATEDEFINITION - Picks a StateDefinition
  - STYLEDEFINITION - Picks a StyleDefinition
  - RENDERERWITHSTATE - Displays a dialog and allows you to select a renderer and formatting. Note: You can set a default style by entering the string with the default style name in the defaultValue. When your binding changes, you should reset it to the default value as in the code below:

```
this.afterAddBindingSource = function (bindingInfo) {
 if(bindingInfo['targetProperty'] === 'Data') {
  this.resetPropertyToDefaultValue('ValueFormat');
 }
};
```

  - STATEFORMATTING - Displays a dialog and allows you to pick a fixed style or state-based style. Note: You can set a default style by entering the string with the default style name in the defaultValue. When your binding changes, you should reset it to the default value as shown in the code above for RENDERERWITHSTATE.
  - VOCABULARYNAME will just pick a DataTags vocabulary at the moment
- mustImplement - if the baseType is THINGNAME, and you specify “mustImplement”, the Mashup Builder will restrict to popups implementing the specified EntityType and EntityName [by calling QueryImplementingThings against said EntityType and EntityName].

  For example:

```
'baseType': 'THINGNAME',
'mustImplement': {
 'EntityType': 'ThingShapes',
 'EntityName': 'Blog'
}
```

- baseTypeInfotableProperty - if baseType is RENDERERWITHFORMAT, baseTypeInfotableProperty specifies which property’s infotable is used for configuration

- sourcePropertyName - when the property's baseType is 'FIELDNAME', this attribute is used to determine which INFOTABLE's fields are to be used to populate the FIELDNAME dropdown list.
- baseTypeRestriction - when specified, this value is used to restrict the fields available in the FIELDNAME dropdown list.
- tagType - if the baseType is 'TAGS' this can be 'DataTags' (default) or 'ModelTags'.
- defaultValue - default undefined; used only for 'property' type
- isBindingSource - true or false; allows the property to be a data binding source, default to false
- isBindingTarget - true or false; allows the property to be a data binding target, default to false
- isEditable - true or false; controls whether the property can be edited in the Composer, default to true
- isVisible - true or false; controls whether the property is visible in the properties window, default to true
- isLocalizable - true or false; only important if baseType is 'STRING' - controls whether the property can be localized or not.
- selectOptions - an array of value / (display) text structures

  For example:
  ```
  [{value: 'optionValue1', text: 'optionText1'},
  {value: 'optionValue2', text: 'optionText2'}]
  ```
- warnIfNotBoundAsSource - true or false; if true, then the property will be checked by the Composer for whether it's bound and generate a to-do item when it's not
- warnIfNotBoundAsTarget - true or false; if true, then the property will be checked by the Composer for whether it's bound and generate a to-do item when it's not

- afterLoad() [optional] - called after your object is loaded and properties have been restored from the file, but before your object has been rendered
- renderHtml() [required] - returns HTML fragment that the Composer will place in the screen; the widget's content container (e.g. div) must have a 'widget-content' class specified, after this container element is appended to the DOM, it becomes accessible via jqElement and its DOM element id will be available in jqElementId
- widgetEvents() [optional] - a collection of events; each event can have the following properties:

- warnIfNotBound - true or false; if true, then the property will be checked by the Composer for whether it's bound and generate a to-do item when it's not
- widgetServices() [optional] - a collection of services; each service can have the following properties:
- warnIfNotBound - true or false; if true, then the property will be checked by the Composer for whether it's bound and generate a to-do item when it's not
- afterRender() [optional] - called after we insert your html fragment into the dom
- beforeDestroy() [optional] - called right before the widget's DOM element gets removed and the widget is detached from its parent widget and dellocated; this is the place to perform any clean-up of resources (e.g. plugins, event handlers) acquired throughout the lifetime of the widget
- beforeSetProperty(name,value) [optional] [Mashup Builder only - not at runtime] - called before any property is updated within the Composer, this is a good place to perform any validation on the new property value before it is committed. If a message string is returned, then the message will be displayed to the user, and the new property value will not be committed. If nothing is returned, then the value is assumed valid.
- afterSetProperty(name,value) [optional] [Mashup Builder only - not at runtime] - called after any property is updated within the Composer. Return true to have the widget re-rendered in the Composer
- afterAddBindingSource(bindingInfo) [optional] - whenever data is bound to your widget, you will called back with this (if you implement it … it's optional). The only field in bindingInfo is targetProperty which is the propertyName that was just bound
- validate() [optional] - called when the Composer refreshes its to-do list. The call must return an array of result object with *severity* (optional and not implemented) and *message* (required) properties. The message text may contain one or more pre-defined tokens, such as {target-id}, which will get replaced with a hyperlink that allows the user to navigate/select the specific widget that generated the message. For example:

```
this.validate = function () {
 var result = [];
 var srcUrl = this.getProperty('SourceURL');
 if (srcUrl === '' || srcUrl === undefined) {
  result.push({ severity: 'warning',
   message: 'SourceURL is not defined for {target-id}'});
 }
 return result;
}
```

Widget API: Runtime

Widget Lifecycle in Runtime

- When a widget is first created, the runtime will obtain any declared properties by calling the runtimeProperties() function. See Callbacks from Runtime to Your Widget for more information.
- The property values that were saved in the mashup definition will be loaded into your object without your code being called in any way.
- After your widget is loaded but before it's put on to the screen, the runtime will call renderHtml() where you return the HTML for your object. The runtime will render that HTML into the appropriate place in the DOM
- Immediately after that HTML is added to the DOM, you will be called with afterRender(). This is the time to do the various jQuery bindings (if you need any). It is only at this point that you can reference the actual DOM elements and you should only do this using code such as:

  ```
  // note that this is a jQuery object
  var widgetElement = this.domElement;
  ```

  This is important because the runtime actually changes your DOM element ID and you should never rely on any id other than the id returned from this. domElementId)
- If you have defined an event that can be bound, whenever that event happens you should call the following:

  ```
  var widgetElement = this.domElement;
  // change 'Clicked' to be whatever your event name is that
  // you defined in your runtimeProperties that people bind to
  widgetElement.triggerHandler('Clicked');
  ```
- If you have any properties bound as data targets, you will be called with updateProperty(). You are expected to update the DOM directly if the changed property affects the DOM - which is likely, otherwise why would the data be bound
- If you have properties that are defined as data sources and they're bound you can be called with getProperty_{propertyName}() … if you don't define this function, the runtime will simply get the value from the property bag.

Runtime APIs Available to Widgets

The following APIs can be accessed by a widget in the context of the runtime:

- this.jqElementId

  This is the DOM element ID of your object after renderHtml().
- this.jqElement

This is the jquery element

- this.getProperty(name)
- this.setProperty(name,value)
- this.updateSelection(propertyName,selectedRowIndices)

  Call this anytime your widget changes selected rows on data bound to a certain propertyName. For example, in a callback you have for an event like onSelectStateChanged(), you'd call this API and the system will update any other widgets relying on selected rows.

Callbacks from Runtime to Your Widget

The following functions on the widget are called by the runtime.

- runtimeProperties() - [optional] Returns a JSON structure defining the properties of this widget

  Optional properties are:

  - isContainer - true or false (default to false); controls whether an instance of this widget can be a container for other widget instances
  - needsDataLoadingAndError - true or false (defaults to false) - set to true if you want your widget to display the standard 25% opacity when no data has been received and turn red when there is an error retrieving data
  - borderWidth - if your widget provides a border, set this to the width of the border. This helps ensure pixel-perfect WYSIWG between builder and runtime
  - supportsAutoResize- if your widget supports auto-resize, set this to true
  - propertyAttributes – if you have STRING properties that are localizable, please list them here. For example, if TooltipLabel1 is localizable:

```
this.runtimeProperties = function () {
return {
 'needsDataLoadingAndError': true,
 'propertyAttributes': {
  'TooltipLabel1': {'isLocalizable': true} }
  }
};
```

- renderHtml() [required] - returns HTML fragment that the runtime will place in the screen; the widget's content container (e.g. div) must have a 'widget-content' class specified, after this container element is appended to the DOM, it becomes accessible via jqElement and its DOM element id will be available in jqElementId

- afterRender() [optional] - called after the widget html fragment is inserted into the dom. Use this.domElementId to find the DOM element ID. Use this. jqElement to use the jQuery reference to this dom element
- beforeDestroy() [optional but highly recommended] - this is called anytime the widget is unloaded, this is the spot to...
  - unbind any bindings
  - clear any data set with .data()
  - destroy any third party libraries or plugins, call their destructors, etc.
  - free any memory you allocated or are holding on to in closures, by setting the variables to *null*
  - there is no need to destroy the DOM elements inside the widget, they will be destroyed for you by the runtime
- resize(width,height) [optional – only useful if you declare supportsAutoResize: true] - this is called anytime your widget is resized. Some widgets don't need to handle this. For example, if the widget's elements and CSS auto-scale. But others (most widgets) need to actually do something based on the widget changing size.
- handleSelectionUpdate(propertyName, selectedRows, selectedRowIndices) - called whenever selectedRows has been modified by the data source you're bound to on that (PropertyName. selectedRows is an array of the actual data and selectedRowIndices is an array of the indices of the selected rows.

> **Note**
>
> To get the full selectedRows event functionality without having to bind a list or grid widget, this function must be defined.

- serviceInvoked(serviceName) - serviceInvoked() is called whenever a service you defined is triggered.
- updateProperty(updatePropertyInfo) - updatePropertyInfo is an object with the following JSON structure{

```
{
DataShape: metadata for the rows returned
ActualDataRows: actual Data Rows
SourceProperty: SourceProperty
TargetProperty: TargetProperty
RawSinglePropertyValue: value of SourceProperty
  in the first row of ActualDataRows
SinglePropertyValue: value of SourceProperty
  in the first row of ActualDataRows
  converted to the defined baseType of the
```

```
  target property [not implemented yet],
SelectedRowIndices: an array of selected row indices
IsBoundToSelectedRows: a Boolean letting you know if this
  is bound to SelectedRows
}
```

For each data binding, your widget's updateProperty() will be called each time the source data is changed. You need to check updatePropertyInfo.TargetProperty to determine what aspect of your widget should be updated. Here is an example from thingworx.widget.image.js:

```
this.updateProperty = function (updatePropertyInfo) {

  // get the img inside our widget in the DOM
  var widgetElement = this.jqElement.find('img');

  // if we're bound to a field in selected rows
  // and there are no selected rows, we'd overwrite the
  // default value if we didn't check here
  if (updatePropertyInfo.RawSinglePropertyValue !==
    undefined) {

  // see which TargetProperty is updated
  if (updatePropertyInfo.TargetProperty === 'sourceurl') {

  // SourceUrl updated - update the <img src=
  this.setProperty('sourceurl',
    updatePropertyInfo.SinglePropertyValue);
  widgetElement.attr("src",
    updatePropertyInfo.SinglePropertyValue);

  } else if (updatePropertyInfo.TargetProperty ===
     'alternatetext') {

    // AlternateText updated - update the <img alt=
    this.setProperty('alternatetext',
      updatePropertyInfo.SinglePropertyValue);
    widgetElement.attr("alt",
      updatePropertyInfo.SinglePropertyValue);
   }
 }
};
```

Note that we set a local copy of the property in our widget object as well so that if that property is bound as a data source for a parameter to a service call (or any other binding) - the runtime system can simply get the property from the property bag. Alternatively, we could supply a custom getProperty_{propertyName} method and store the value some other way.

- getProperty_{propertyName}() - anytime that the runtime needs a property value, it checks to see if your widget implements a function to override and

get the value of that property. This is used when the runtime is pulling data from your widget to populate parameters for a service call.

Tips

- Use this.jqElement to limit your element selections. This will reduce the chance of introducing unwanted behaviors in the application when there might be duplicate IDs and/or classes in the DOM.
  - Don’t do the following: `$('.add-btn').click(function(e){...do something...});`
  - Do this:
    ```
    this.jqElement.find('.add-btn').click(function(e){
    ...do something...});
    ```
- Logging - we recommend that you use the following methods to log in the Mashup Builder and Runtime environment:
  - `TW.log.trace(message[, message2, ... ][, exception])`
  - `TW.log.debug(message[, message2, ... ][, exception])`
  - `TW.log.info(message[, message2, ... ][, exception])`
  - `TW.log.warn(message[, message2, ... ][, exception])`
  - `TW.log.error(message[, message2, ... ][, exception])`
  - `TW.log.fatal(message[, message2, ... ][, exception])`

  You can view the log messages in the Mashup Builder by opening the log window via the Help>Log menu item; in the Mashup runtime, you can now click on the "Show Log" button on the top left corner of the page to show log window. If the browser you use supports console.log(), then the messages will also appear in the debugger console.
- Formatting - if you have a property with baseType of STYLEDEFINITION, you can get the style information by calling
  ```
  var formatResult = TW.getStyleFromStyleDefinition(
   widgetProperties['PropertyName']);
  ```
  If you have a property of baseType of STATEFORMATTING:
  ```
  var formatResult = TW.getStyleFromStateFormatting({
    DataRow: row,
    StateFormatting: thisWidget.properties['PropertyName']
  });
  ```
  In both cases formatResult is an object with the following defaults:
  ```
  {
     image: '',
     backgroundColor: '',
     foregroundColor: '',
  ```

```
    fontEmphasisBold: false,
    fontEmphasisItalic: false,
    fontEmphasisUnderline: false,
    displayString: '',
    lineThickness: 1,
    lineStyle: 'solid',
    lineColor: '',
    secondaryBackgroundColor: '',
    textSize: 'normal'
};
```