Application Development Guide for ThingWorx 8

September 2019

Contents

About this Guide

The *ThingWorx Application Development Guide* describes the best practices for building applications on the ThingWorx platform.

IoT applications are built and deployed as extensions on the ThingWorx platform. Extensions enable you to package and version your application entities, custom widgets, and so on, which further helps in deploying and upgrading your applications.

Prerequisites

To develop ThingWorx applications, you must have basic knowledge of:

- ThingWorx Composer and its modeling capabilities
- ThingWorx Mashup Builder

Optional Installations for Java Services

If you are creating an extension that has services implemented in Java, you must have the following installed:

- Java SE Development Kit (JDK) 8
- ThingWorx Extension SDK for your version of ThingWorx
- Eclipse IDE for Java EE developers
- Eclipse plugin for ThingWorx extensions, which is available on PTC Marketplace

1

Tips and Tricks: Getting Started with Building ThingWorx Applications

ThingWorx is a model-based application development platform. With the ThingWorx Platform, you can build and deploy IoT solutions to address your business problems and optimize processes. You can build a customized user interface for your business requirements and provide a real-time digital view of a remote physical object to help optimize operational processes.

An application in ThingWorx is a collection of entities that are packaged as one or more extensions.

This guide does not provide a detailed explanation on ThingWorx basics, but rather suggests best practices for using ThingWorx to build applications. Use this guide along with other resources such as ThingWorx Help Center, ThingWorx Extension Development Guide, ThingWorx Developer portal, and so on.

The following sections provide a summary of the best practices for developing applications in ThingWorx Composer. The goal of these sections is to provide an overview and overall guidance for building applications without detailing specific details. For more in-depth learning, you are referred to the relevant topics in this help center. It is recommended that you review these sections before you start building an application in this help center.

It is recommended that you review these best practices before you start building an application. This will ensure that you take full advantage of the PTC IoT technology to create scalable and powerful applications for your customers.

Building a ThingWorx Application

The recommended workflow for building an IoT application on ThingWorx Platform is as follows:

1. Create a Project.
2. Create Model Tags.
3. Create Thing Shapes to implement common or shared properties and services.

Note

It is recommended that you use Thing Shapes to define the properties and services. If you define properties and services on a Thing Template, it is difficult to move their definitions to a Thing Shape.

4. Create Thing Templates that implement one or more Thing Shapes.
5. Decide how you will store data, for example, using Value Streams, Data Tables, and so on.
6. If general services for business logic are required, consider creating manager or helper things to consolidate the services.
7. Create the user interface for the application in the Mashup Builder.
8. Set up visibility and permissions for Organizations, User Groups and Users.
9. Package your application as an extension. If your application has dependencies on other extensions, define them in the metadata of the extension. See the section Installing a ThingWorx Application on page 8 to install the extension.

Installing a ThingWorx Application

To install the IoT application that is packaged as an extension, the recommended workflow is as follows:

1. Install the dependent extensions, if any.
2. Install the application extensions.
3. Create Thing Templates that derive from the Thing Templates of the extension.
4. Create Remote Things to represent the connected assets.

Best Practices at a Glance for Building ThingWorx Applications

Consider the following best practices when you build your IoT application:

- Include entities in a Project. Use only one Project for one application.
- Tag entities with Model Tags.
- Make all extension entities non-editable.
- Create localization tokens for all user interface labels.
- Use JavaScript for implementing services.
- Use Thing Shapes to define services and properties.
- Use a unique namespace prefix for entity names, properties, and services.
- Define Organizations and User Groups for entities to set visibility and permissions.
- Split large applications and solutions into smaller extensions.
- Do not use fixed asset Things to execute services in Mashups. Asset Things should be dynamically selected or looked up, and then their services should be executed.

Note

To avoid problematic scenarios, use the services of helper or manager Things. These services take the name of an asset Thing as the input parameter that is later used in Mashups.

Additional Resources for Getting Started with ThingWorx

This section lists the additional resources that you can reference for a better understanding of the concepts.

Additional Resources for More In-Depth Learning

For more in-depth information on	Link	Resource
ThingWorx Help Center	http://support.ptc.com/help/thingworx_hc/thingworx_8_hc/en/index.html	Help Center
Getting started with ThingWorx	http://support.ptc.com/help/thingworx_hc/	Help Center

For more in-depth information on	Link	Resource
	thingworx_8_hc/en/index.html#page/ThingWorx%2FHelp%2FGetting_Started%2FGettingStarted.html%23	
ThingWorx video tutorials	http://support.ptc.com/help/thingworx_hc/thingworx_8_hc/en/index.html#page/ThingWorx%2FHelp%2FGetting_Started%2FThingWorxVideoTutorials%2FThingWorxVideoTutorials.html%23	Help Center
ThingWorx Developer Portal	https://developer.thingworx.com/en	Developer portal
Application development tips and tricks	https://developer.thingworx.com/resources/guides/application-development-tools-tips-tricks	Developer portal
Object-oriented User Interface design tips	https://developer.thingworx.com/resources/guides/reusable-components-how-	Developer portal

2

Modeling your Assets

This section provides a summary of the best practices for modeling your assets.

Modeling Assets with Things, Thing Templates, and Thing Shapes

Things are representations of physical devices, assets, products, systems, people, or processes that have properties and business logic.

With the purpose of defining one behavior once and reusing it across your application, ThingWorx facilitates modeling your application by providing Thing Templates and Thing Shapes. Thing Templates provide the base functionality to Things with their properties, services, events, and subscriptions. Thing Shapes allow a reuse of properties across assets in a single application.

Things inherit functionality from Thing Templates. A Thing Template can derive one or more additional characteristics by implementing Thing Shapes.

End users interact with Things for information in applications and for reading or writing data.

Best Practices for Creating Things, Thing Templates, and Thing Shapes

Use the following best practices while creating Things, Thing Templates, and Thing Shapes:

- Define naming conventions for your Things, Thing Templates, and Thing Shapes. Keep the following points in mind:
    - Use a standard nomenclature across entities. For example: You could use a unique namespace prefix across different entity types to avoid conflicts.
    - Provide meaningful names to the entities.
    - Try to include a good description for every entity that you create.
    - Avoid ambiguous names.
    - Avoid long entity names wherever possible.

  See the section on naming entities in the ThingWorx Help Center for more information.
- Use tags for all the entities that you create and modify. For complex applications, this helps in differentiating between various modules of the same application or multiple applications. For customers, it helps to differentiate between entities of different applications on the same instance of ThingWorx platform. As a developer, it permits you to quickly filter through the entities in ThingWorx Composer and export a module of the application for testing purposes. It is recommended to set up naming conventions for tags, especially if you plan to use different tags across different modules of the same application.
- Group the same entities in a module into a project. Each module has its own project.

- Whenever possible, use Thing Shapes.

Note

It is recommended that you use Thing Shapes to define properties and services. If you define properties and services on a Thing Template, it is difficult to move their definitions to a Thing Shape.

  - Define all properties, services, and events at a Thing Shape level.
  - A Thing Template is used to group Thing Shapes and to support inheritance.
  - Whenever possible, do not implement properties or services on Thing Templates and Things.
  - Allow service overrides on Thing Shapes and Thing Templates.
  - Ensure that out-of-the-box Thing Shapes and Thing Templates are non-editable.
  - Define annotations to identify supported APIs. This ensures that services and properties are maintained on upgrade.
- Use a Thing Shape to encapsulate specific functionality. By using Thing Shapes, you can easily add new functionality to existing Thing Templates in ThingWorx. Do not implement Thing Shapes in the Thing directly. Inherit the characteristics of the Thing Shape in the Thing through the Thing Template.

  For example, if you have implemented your assets using custom Thing Templates and Things, it is simple to add a Thing Shape to those Thing Templates. However, if the properties or services were defined directly on a Thing Template, you will need to recreate all your assets to use the new functionality, because the base Thing Template cannot be changed for a Thing Template or a Thing.
- Do not derive Things directly from system Thing Templates. It is recommended that you create custom Thing Templates that derive from the base Thing Templates provided by ThingWorx. This allows you to add additional functionality later by implementing additional Thing Shapes and adding them to your Thing Templates.
- If your application has many connected remote devices or if it uses the ThingWorx Edge Micro Server or the EDGE SDKs, use the `RemoteThing` Thing Template instead of the `GenericThing` Thing Template as the base Thing Template. Depending on whether you need file transfer or tunneling, use the `RemoteThingWithTunnels`, `RemoteThingWithFileTransfer`, or `RemoteThingWithTunnelsAndFileTransfer` Thing Template. It is recommended to automate the process of adding remote bindings through

scripts that check if unbound remote Things exist, and create them automatically on the platform. This enables you to automatically have Things created on the platform, as new devices are shipped and go into production.

Note

You cannot change the base Thing Template after you create a Thing or Thing Template.

- Use networks to define relationships between your Things. In a network, a Thing can be the parent, child, or sibling of another Thing. This enables you to model a hierarchical structure.

Sustainability Considerations

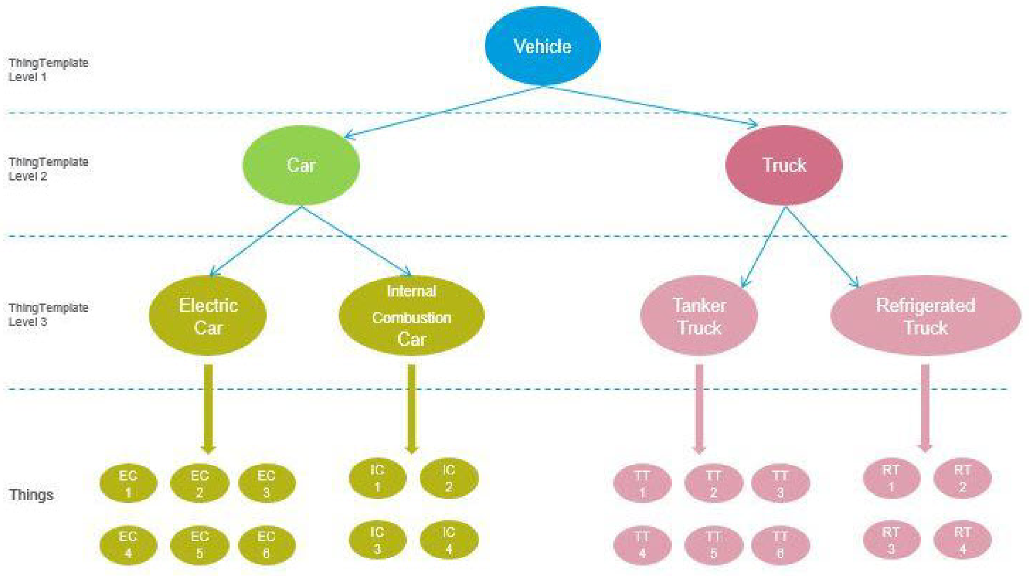
This section describes other considerations as applicable:

- Security
    - Ensure that you make all the Things, Thing Templates, and Thing Shapes non-editable.
    - Add appropriate visibility, runtime and design time permissions on the Thing Templates or Thing Shapes.
    - Define User Organizations, User Groups, and Users that have access to the Thing Shapes, Thing Templates, or Things.
- Upgrades
    - Organize Things, Thing Templates, and Thing Shapes using projects and tags. This allows you to perform tasks such as searching, exporting, modifying visibility permissions, and identifying entities for an upgrade.
    - Ensure that you use the right naming convention while creating entities. This saves costs of renaming entities during upgrades.
- Extension and Customization
    - To extend component Things, create subtype Thing Templates that inherit the OOTB Thing Templates.
    - Override services and properties in the subtype Thing Templates as instructed by the extension developer.
    - Introduce new services and properties in new Thing Shapes.

Example: Using Things, Thing Templates or Thing Shapes

When do you use Thing Templates?

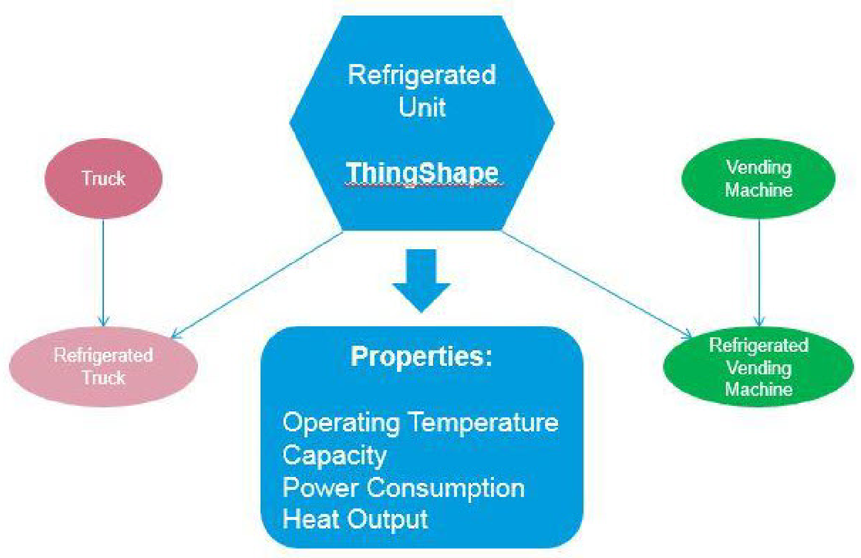
For example: If your company rents smart connected vehicles, you can create a `Vehicle` Thing Template by using the `GenericThing` Thing Template as the base Template. Based on the `Vehicle` Thing Template, create `Car` and `Truck` Thing Templates. Based on the `Car` Template, create an `Electric Car` Thing Template and an `Internal Combustion Car` Thing Template. Based on the `Truck` Thing Template, create a `Tanker Truck` Thing Template and a `Refrigerated Truck` Thing Template. Create individual Things that implement these Thing Templates. The following image illustrates this scenario:



When do you use Thing Shapes?

Thing Templates can implement 0, one or multiple Thing Shapes.

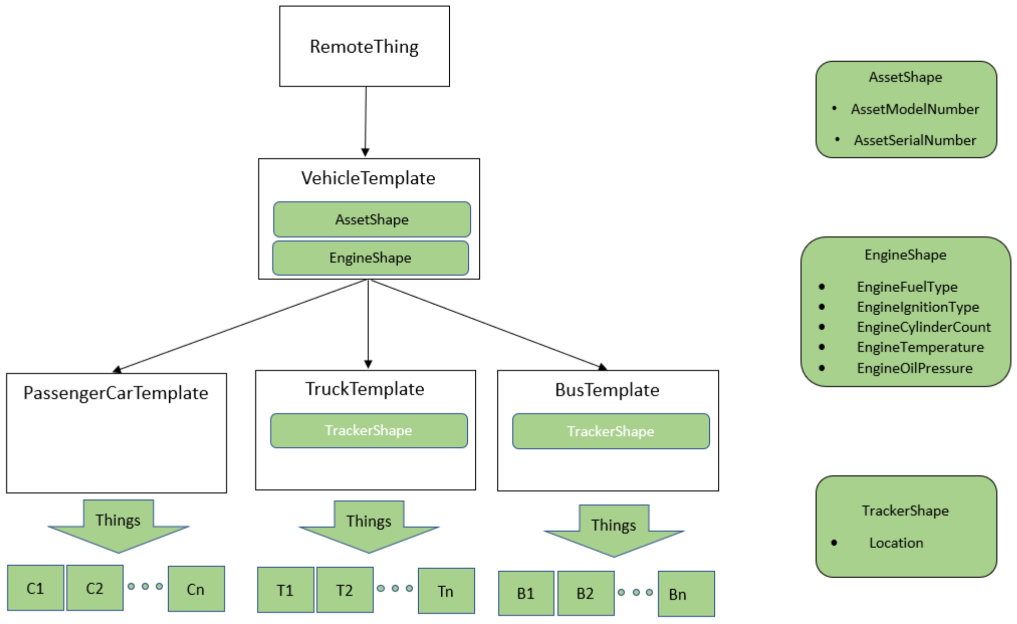
A company owns refrigerated trucks (based on a `Truck` Thing Template) and refrigerated vending machines (based on a `Vending Machine` Thing Template). These two entities have common properties such as operating temperature, capacity, power consumption, and heat output. Instead of defining this behavior twice, it is efficient to define it in a `Refrigerated Unit` Thing Shape. This Thing Shape can be reused by both entities.



How do Things, Thing Templates, and Thing Shapes work together?

The following image depicts how Thing Templates, Thing Shapes, and Things work together. Note the following points:

- The `VehicleTemplate` Thing Template implements the system Thing Template, `RemoteThing` as the base template, and inherits the `AssetShape` and `EngineShape` Thing Shapes. As a result, the `VehicleTemplate` has all properties and services of the `RemoteThing` Thing Template and the `AssetShape` and `EngineShape` Thing Shapes.
- The `PassengerCarTemplate`, `TruckTemplate`, and `BusTemplate` Thing Templates implement the `VehicleTemplate` Thing Template as the base template. As a result, they inherit all properties and services of the `VehicleTemplate` Thing Template.
- The `TruckTemplate` and `BusTemplate` Thing Templates inherit the `TrackerShape` Thing Shape. In addition to the properties of the `VehicleTemplate` Thing Template, they have properties of the `TrackerShape` Thing Shape.
- Things `C1`, `C2`, …., `Cn` implement the `PassengerCarTemplate` as the base template.
- Things `T1`, `T2`, …., `Tn` implement the `TruckTemplate` as the base template.
- Things `B1`, `B2`, …., `Bn` implement the `BusTemplate` as the base template.



Additional Resources for Things, Thing Templates, and Thing Shapes

This section lists the additional resources that you can reference for a better understanding of the concepts.

Additional Resources for More In-Depth Learning

For more in-depth information on	Link	Resource
ThingWorx Model Definition and Composer	http://support.ptc.com/help/thingworx_hc/thingworx_8_hc/en/#page/ThingWorx%2FHelp%2FComposer%2FModeling.html%23	Help Center
Naming Entities in ThingWorx	http://support.ptc.com/help/thingworx_hc/thingworx_8_hc/en/#page/ThingWorx%2FHelp%2FComposer%2FNamingEntities.html	Help Center

Storing Data in ThingWorx

ThingWorx provides entities and methods to store data. You can store data in Data Tables, Thing properties, Streams, Value Streams, and configuration tables.

While developing your application, you need to consider how ThingWorx handles data storage. Choosing the correct data storage is very important since it affects the outcome of the project, its scalability and reliability, and the user experience.

The following section describes the storage options in the ThingWorx model:

Data Table

- Use for less than 100,000 rows of data.
- Use for static datasets and static lookup tables. For highly dynamic and larger datasets, use a relational database that is connected via a `Database` Thing Template.

  Note

  Use a relational database for complex queries and joins.

- Use for key-based queries and storage as well as for easily enabling updates and deletes based on the primary key.

  For example: You can store information about the inventory of a smart connected vending machine, where each position in the inventory is a primary key. You can also use Data Tables to store information about the irrigation programs that are available for a crop management device, where each irrigation program is a row with a primary key.
- To manipulate or query the data row-by-row, use Data Tables.
- Use indexes when working with Data Tables.

Note

Data Tables do not support high-speed writing, as they do not have queuing mechanism like Streams and Value Streams.

Thing Property

Use Thing properties to store data about a Thing in ThingWorx. Properties have the following data storage options:

- Read-only - Use the read-only option for static values that should not be changed at runtime. However, if you want, you can change the default value.

- Persisted - Use the persisted option if you want the value of the property to be saved even after a ThingWorx server restart, and if the value of the property can be changed at runtime.
- Logged - Use the logged option for properties whose values update continuously. This is time series data that can be stored in Value Streams.

Note

Do not use explicit properties to store historical data. Instead use Streams or Value Streams.

Stream

Use for logging time-driven process events or activities your devices.

For example, create a stream to log issues about your device activities, or to record when your device disconnects from and reconnects to the ThingWorx platform. Streams are optimized for high-speed writing, and they have a configurable cache system.

Value Stream

Use for storing time series data that is obtained from the properties of a thing.

With streams, Data Tables are created. When you use Value Streams, the creation of sparsely populated Data Tables is eliminated.

Thing-centric access of data in a Value Stream provides in-built support for multi-tenancy.

If you are using an RDMS (PostgreSQL, MSSQL, H2), all records, even from different Value Streams for different Things, are written to the same table in the database.

If you use PostgreSQL, in the table for ValueStream in the PostgreSQL database, each row holds the record for only one property. This means that the Value Stream tracks the value change of each property independently. After using the `QueryPropertyHistory` service, it checks the data flow for each property in the Thing and collects all latest updates (each has a different update time) into one infotable result.

The following table provides information about the key differences between Streams and Value Streams. Use this information to decide on the type of entity to use to store time series data in your application:

Streams	Value Streams
Streams can store any type of time series data.	Value Streams can store time series data from the property of a Thing. Value streams are bound to the properties of a Thing.
You can query data from Streams directly by using their own services. The result of the query is the entire row of data.	You cannot query data from Value Streams directly. Instead, use services defined on the Thing to query data from the Value Stream. For example: `QueryPropertyHistory`
To add a row of data to a Stream, use the `WritePropertiesToStream` service.	To add data to a Value Stream, select the **Logged** check box for a property.
Streams can store contextual data. For example, whenever a specific event is triggered, you can add the values of the other properties. This helps in analysis of data.	Value Streams cannot store contextual data.

When to use Streams or Value Streams?

- Use Value Streams and Streams to store and retrieve time series data. Depending on the amount of data that you need to store, choose the correct data storage option.
- Use Streams when you want to query data only within small time periods.
- Split Things across several Value Streams to improve index performance.

Configuration Table

Use to build customizable applications that can be safely upgraded on the ThingWorx platform.

You can add a configuration table on a Mashup entity based on a pre-defined data shape. This simplifies the process of building extension Mashups that can be customized, while also still supporting in-place upgrades, because configuration table values are carried forward during extension upgrades.

Best Practices for Handling Data-Centric Modeling

Use the following general best practices to handle data-centric modeling in ThingWorx:

- If you have data that will not change or will be overwritten the next time it is changed/loaded, and it is associated with a Thing, create an infotable property for that Thing and assign a proper data shape. In this way, you can access the data through the Thing. You can also use configuration tables, or for larger amounts of data, use a data table.f
- Use data caching as much as possible.

  For example, instead of querying the database on each `DataChange` event, implement a cache, as an infotable, that is refreshed at set intervals.
- Archive the data that you no longer need.

  While designing your application, you must decide what data is frequently used. You can store this data in the application database. Move the old data as soon as possible to an external server, such as a ThingWorx federated instance or a database server.
- Provide start and end date parameters to the query methods to limit the amount of data that the query retrieves. This reduces the processing time and improves performance.
- For high-volume data ingestion scenarios (over and above the ingestion rates outlined in the ThingWorx Platform Sizing Guide), consider creating multiple Persistence Providers that connect to separate database instances. This ensures that data goes into different tables in the database. If you add multiple databases, the Persistence Providers can point to specific databases. In this case, you need data migration.
- Ensure that your Data Tables have less than 100,000 rows.
- Querying data from Data Tables and Streams should only take a few seconds. If these Data Tables and Streams have more than 100,000 rows, the queries perform slowly.
- Ensure that you plan how you intend to purge your old data. Purging data is important, as it helps improve the performance of an application.

Additional Resources for Data

This section lists the additional resources that you can reference for a better understanding of the concepts.

Additional Resources for More In-Depth Learning

For more in-depth information on	Link	Resource
Data Storage options	http://support.ptc.com/help/thingworx_hc/thingworx_8_hc/en/#page/ThingWorx%2FHelp%2FComposer%2FDataStorage%2FDataTables%2FDataTables.html%23	ThingWorx Help Center
Data Storage and Data Modeling	https://www.ptc.com/support/-/media/168BEE102EA6474C9F-DD6E7FDDA8F259.pdf?sc_lang=en	ThingWorx Model and Data Best Practices guide

Determining the Correct Data Store Option

Determining the data store option is specific to your application. You may have to consider the process and cost of the data store option that you choose for your application. There are multiple data store options available:

- The internal database that ThingWorx uses, for example, PostgreSQL, MS SQL, or SAP HANA. Usually, PostgreSQL performs well up to an estimated 500 GB of data storage.

  Note

  H2 is only supported for non-production instances because they do not support large amount of data.

- Use DataStax Enterprise (DSE) when the writes per second exceeds the limitations of the internal database that ThingWorx uses. Refer to the ThingWorx sizing calculator. If you have completed sizing of your ThingWorx server, you can review it here.

Note

As of version 8.5.0 of ThingWorx platform, DSE is no longer for sale and will not be supported in a future release. Reference the End of Sale article for more information.

- Use any data storage server that provides a JDBC connector. The ThingWorx platform can use JDBC drivers to connect to a relational database such as Oracle.

Note

Persistence Providers enable the connection to a data store and the ability to perform a CRUD operation on that data. Persistence providers are an entity type in ThingWorx. For more information about Persistence Providers, see ThingWorx Model and Data Best Practices guide.

The following table provides recommendations for the data store to use depending on the type of data in your application:

Data Store Option	Type of Data
Relational Database	• Data that uses complex SQL such as joins, cursors, temporary tables, and so on. • For large data sets of non-time series data that has a loose relationship between the data that needs to be used and executed on the database server.
ThingWorx platform data storage, such as PostgreSQL, MS SQL, or SAP HANA	Simple historical data, time series data, or data that does not require compounding or recursive calculations.
DataStax Enterprise **Note** As of version 8.5.0 of ThingWorx platform, DSE is no longer for sale and will not be supported in a future release. Reference the End of Sale article for more information.	For large data sets of non-time series data that has sequential data and does not need the data to be executed on the database server.

Best Practices for Choosing Your Data Store Option

Use the following best practices to choose your data store option:

- Use an external data store option if you need to perform any of the following tasks:
  - Query Streams or Value Streams for a large amount of data
  - Query results with millions of rows
  - Implement database-level custom logic and calculations, for example: stored procedures, triggers, joins, relational data queries, and so on.
  - Use ThingWorx as a data lake
  - Integrate with third-party analytics and reporting tools
  - Add additional indices and database infrastructure

  Note

  It is recommended that you archive historical data that is not frequently accessed.

- Decide how much data you need to store. If data can be stored in small data sets (around 100,000 records or less), use the ThingWorx platform storage.
- Decide what kind of data your application has.
- How frequently you query your data affects the performance of the application. Depending on how frequently you want to query your data, it is recommended to use either of the following options:
  - To query data frequently, use a JDBC connector to query the database. You can use this JDBC connector with PostgreSQL or DataStax Enterprise.

    Note

    As of version 8.5.0 of ThingWorx platform, DSE is no longer for sale and will not be supported in a future release. Reference the End of Sale article for more information.

  - If you do not need to retrieve historical data very frequently, you can move all data to a separate ThingWorx instance (in a Data Table or Stream) or to a database server.

Creating, Implementing, and Testing Services

Thing services are functions that a Thing can perform. Services are used internally in the ThingWorx platform and by Mashups, and they can be reached from any external source with appropriate access. You can define services on the following ThingWorx entities only:

- Things
- Thing Templates
- Thing Shapes
- Resources
- Authenticators

Services are implemented through JavaScript, SQL, or Java.

Note

Java is available for extensions only.

A service can be invoked through a URL, a REST client application, or by another service in ThingWorx.

Create custom services for your model in ThingWorx to meet the requirements of your project.

Best Practices for Creating and Implementing Services

Use the following best practices while creating and implementing services:

- Define naming conventions for your services. Keep the following points in mind:
  - Provide a logical, meaningful name and description to the service.
  - Use a standard nomenclature across services. For example, prefix the service name with a verb and a description of the service. Some examples of verbs are below:

Verb	Description
Get	Retrieve values from the database.
Set	Set the value of database entries.
Query	Return a group of records from the database.
Add	Add records to the database.
Update	Update records in the database.
Delete	Delete records in the database.

Verb	Description
Validate	Validate records in the database.
Purge	Purge records in the database.
Create	Create a set of records in the database.
Import	Import data to the database.
Export	Export data from the database.
Parse	Parse data.

For example: A service that retrieves historical data, `getHistory` is the recommended name, instead of `History`.

- ○ Avoid ambiguous names.
- ○ Avoid long service names wherever possible.

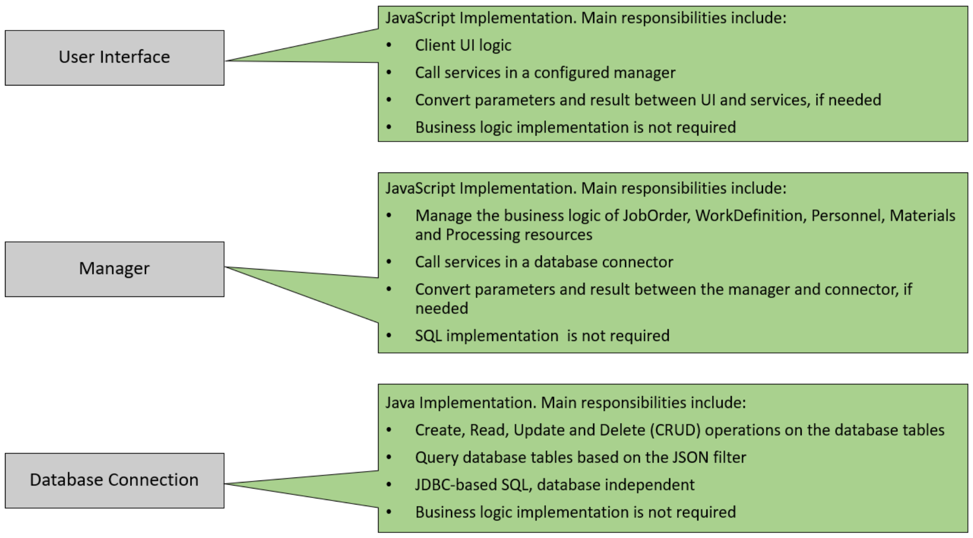
See the section on naming entities in the ThingWorx Help Center for more information.

- Group common properties and services in a single entity, preferably a Thing Shape.
- Whenever possible, implement services on Thing Shapes.

Note

It is recommended that you use Thing Shapes to define properties and services. If you define properties and services on a Thing Template, it is difficult to move their definitions to a Thing Shape.

- Design services for different layers such as a user interface, business logic, and data retrieval. The services in different layers have different responsibilities. The following image provides an example of the responsibilities in different layers:


User Interface
JavaScript Implementation. Main responsibilities include:
• Client UI logic
• Call services in a configured manager
• Convert parameters and result between UI and services, if needed
• Business logic implementation is not required
Manager
JavaScript Implementation. Main responsibilities include:
• Manage the business logic of JobOrder, WorkDefinition, Personnel, Materials and Processing resources
• Call services in a database connector
• Convert parameters and result between the manager and connector, if needed
• SQL implementation is not required
Database Connection
Java Implementation. Main responsibilities include:
• Create, Read, Update and Delete (CRUD) operations on the database tables
• Query database tables based on the JSON filter
• JDBC-based SQL, database independent
• Business logic implementation is not required

- While writing a service, try to reuse the available snippets from the ThingWorx code snippet library. If you are not sure how to use any of the code snippets, test them before using them in your service.
- If you expect your service to take a long time to complete, ensure that the user is not able to trigger the service more than once at the same time.
- If the service is based on an event trigger in a Mashup, use the `ServiceInvokeCompleted` event to allow data flow in an application.
- To refresh data in Mashups, it is recommended to use an Auto Refresh widget or the `GetProperties` service.

  If a Mashup uses the `GetProperties` service and the browser supports WebSockets, then the browser automatically receives values of updated properties in real time from the server. In this case, there is no need to use the Auto Refresh widget. This capability is available only if you select the **Automatically update values when able** check box from the service properties panel.

  If you are using the Auto Refresh widget, PTC recommends setting the Auto Refresh widget to a minimum of 15 seconds to avoid taxing your system.

Note

Do not use a server-side delay service, as it can lead to other services being blocked. This can result in an application crash.

- For simple conversions at runtime, it is recommended to use the Expression widget rather than services.

  For example, if you are displaying the temperature in °C and you want the user to be able to see it in °F, use the Expression widget.
- Add checks to your code to avoid the possibility of the end user receiving an error.

  For example, if your code needs a couple of input parameters to run, create checks to ensure that those input parameters are not null when the service is executed.
- If your application will be localized and if there is text in the UI elements that changes dynamically based on the result of services, ensure that you do not hardcode text values in your services that will be displayed in the Mashup. This is because the result of the services returning the text needs to be localized. This also makes it easier to maintain and modify the UI text in the future.
- While creating custom services, decide which users or user groups should have the permissions to invoke this service. For more information, see the Securing the Applications Built on the ThingWorx Platform Using Visibility and Permissions on page 70.
- A possible result of the execution of services can be the creation of ghost entities. Ghost entities are created dynamically via code/scripting, rather than from the ThingWorx Composer. Creating ghost entities is a bad practice. For more information, see Detection, Removal and Prevention of Ghost Entities in ThingWorx.

  If ghost entities are created, you can eliminate them in one of the following ways:
  - Restart the ThingWorx server. This reloads the JVM memory from the persisted data and excludes ghost entities.
  - Use the Ghost Entities Cleaner extension. You can download it from the PTC Marketplace.
  - Use the "`try-catch`" mechanism to delete ghost entities. For more information, see Example: Creation and Deletion of Ghost Entities on page 30.

- The default script timeout setting on the ThingWorx platform is 30 seconds. If a script runs longer than the default setting, the platform terminates the execution. The ThingWorx administrator can configure the script timeout in the basic settings section of the `platform-settings.json` file.

Best Practices for Testing Services

Use the following best practices while testing services:

- Test your service incrementally, as you build it. Use script logger messages, if appropriate.

> **Note**
>
> Any service with an infinite loop can bring down the ThingWorx server.

- While testing a service, check logs for error messages. This is particularly useful for services in subscriptions. The following log files are useful:
  - `ErrorLog.log` – Provides detailed stack tracing for errors
  - `ScriptErrorLog.log` – Provides service context information (stack error line number)

    Errors are logged in this file only if you select the **Enable Script Stack Tracing** check box in the **Configuration** tab of the **LoggingSubsystem** subsystem.

    > **Note**
    >
    > The above files are available in the `ThingworxStorage/logs` folder. You need access to the ThingWorx server to read these files.

- If the service is called from a Mashup, test it in the Composer and in the Mashup.
- If user inputs are required for the service to execute, you can use the Validator widget.
- Use the Developer Tools that are available in browsers to check the result of a service. This is useful when you want to debug a sequence of services that executes in a Mashup. This tool displays the service results in the context of that specific execution.

Tips

- If you want to search for services, do not open the entity in Edit mode. Use the Preview link to the left of the name of the entity to open the entity in View mode.

Other Considerations

- Security
  - To enhance security, use service overrides to deny permissions on critical services available on the platform.
- Upgrades
  - Use good coding practices and do not create large monolithic services.

Example: Creation and Deletion of Ghost Entities

The following example results in the creation of a Ghost Thing:

```
var params = {
name: "GhostThing" /* STRING */,
description: "Ghost" /* STRING */,
thingTemplateName: "GenericThing" /* THINGTEMPLATENAME */
};

// Successfully creates the GhostThing entity.
Resources["EntityServices"].CreateThing(params);
// Code that will cause an exception to be thrown by the service.
var makeError;
// Exception is thrown as the notThere property does not exist
 on the makeError variable at this point.
makeError.notThere = 1;
```

Use the "`try-catch`" mechanism to delete ghost entities:

```
try {
var params = {
name: "GhostThing" /* STRING */,
description: "Ghost" /* STRING */,
thingTemplateName: "GenericThing" /* THINGTEMPLATENAME */
};
Resources["EntityServices"].CreateThing(params);
// Any code that could potentially cause an exception should
// be included in the try-catch block.

// The following code will cause an exception to be thrown
// as makeError.notThere has not been defined.
var makeError;
makeError.notThere = 1;

} catch (err) {
// If an exception is caught, we need to delete the entity
// that was created to prevent it from becoming a Ghost Entity
var params = {
name: "GhostThing" /* THINGNAME */
};
Resources["EntityServices"].DeleteThing(params);
}
```

Additional Resources for Services

This section lists the additional resources that you can reference for a better understanding of the concepts.

Additional Resources for More In-Depth Learning

For more in-depth information on	Link	Resource
Services	http://support.ptc.com/help/Thingworx_hc/Thingworx_8_hc/en/index.html#page/ThingWorx%2FHelp%2FComposer%2FModeling.html%23	Help Center

Working with Events, Alerts, and Subscriptions

Events are used to indicate a change in condition in the ThingWorx platform. They can be triggered by a condition being met, by calling a service, or by making a REST call. ThingWorx has several standard types of events that are triggered based on certain conditions, for example when the value of a property changes (`DataChange`) or when a Thing is started (`ThingStart`).

An alert is a standard type of event that is based on a property's state. It is automatically triggered when the property value meets a particular condition that has been defined by a user.

A subscription is required to do something when an event, including alerts, is triggered. A subscription is a special service that is executed when the event it subscribes to is triggered.

Events require a predefined data shape. The data shape stores data associated with the event, which can be accessed by a subscription. You can add subscriptions to an entity in an extension to perform custom behavior when an event is fired.

Best Practices for Creating Events, Alerts, and Subscriptions

Use the following best practices while creating events, alerts, and subscriptions:

- Define naming conventions for your events, alerts, and subscriptions. Keep the following points in mind:
  - Use a standard nomenclature across entities. For example: You can use a unique namespace prefix across different entity types to avoid conflicts.
  - Provide meaningful names to the entities.
  - Try to include a good description for every entity that you create.
  - Avoid ambiguous names.
  - Avoid long entity names wherever possible.

  See the section on naming entities in the ThingWorx Help Center for more information.
- Define all events and subscriptions at a Thing Shape level.

Note

It is recommended that you use Thing Shapes to define properties and services. If you define properties and services on a Thing Template, it is difficult to move their definitions to a Thing Shape.

- Define alerts individually for every property of the Thing, Thing Template, or Thing Shape.
- Ensure that events and alerts have a subscription tied to it. This ensures that whenever an event or alert is triggered, functionality is executed in the application. This results in data flow in the application.
- While developing your application, you could execute an automatic backup through the scheduler subscription.
- While testing services that are defined in subscriptions, check the application logs to verify that the subscription is running as expected.

  For information about best practices on creating services for subscriptions, see Services.

Additional Resources for Events, Alerts, and Subscriptions

This section lists the additional resources that you can reference for a better understanding of the concepts.

Additional Resources for More In-Depth Learning

For more in-depth information on	Link	Resource
Events	http://support.ptc.com/help/thingworx_hc/thingworx_8_hc/en/#page/ThingWorx%2FHelp%2FComposer%2FThings%2FThingEvents.html%23	Help Center
Alerts	http://support.ptc.com/help/thingworx_hc/thingworx_8_hc/en/#page/ThingWorx%2FHelp%2FComposer%2FAlerts%2FAlerts.html	Help Center
Subscriptions	http://support.ptc.com/help/thingworx_hc/thingworx_8_hc/en/#page/ThingWorx%2FHelp%2FComposer%2FThings%2FThingSubscriptions%2FThingSubscriptions.html%23	Help Center

3

Designing the User Experience for Your ThingWorx Application

ThingWorx applications are visualized as Web pages. When you design the user interface of your application, think about the following questions:

- Should you use a static or a responsive layout?
- Can the Mashups be reused for a mobile device?
- Can you build the application with existing ThingWorx widgets or do you need to use custom widgets?
- What is the minimum physical screen size for which the application must be designed? Should it be for a 5-inch mobile screen or a 10.4-inch tablet?

A consistent user interface is an important aspect of a good application design. Examples of good design practices include:

- If one list can be double-clicked for more information, then the other lists should have the same functionality.
- Ensure that the common buttons are available in the same place on every page.
- Use similar wording for text and labels.
- Use the same color scheme throughout the application.

What are Mashups and Masters in ThingWorx?

This section explains how to work with Mashups and Masters.

Mashups

Mashups are visualizations for Web pages that display data from a ThingWorx model or devices based on the business and technical requirements of your application.

Use the Mashup Builder to create the user interface for your application. The Mashup Builder combines the data services available in ThingWorx with a set of visualization components, called widgets, to create unique Web pages. The Web pages combine data from multiple sources.

Consider these concepts for building Mashups:

- Widgets are the components that you place on the Mashup. A widget can be a grid or chart for data rendering. Widgets also include basic HTML elements such as text boxes, buttons, and navigation links.
- ThingWorx data services are used to retrieve data that can be used in the Mashup. Visualization widgets are used to display the data on the Mashup.

  Services can also be called in response to other service states and user interaction.
- Widgets support styles and states. For example, a text box supports a style for the font size, font color, and background color, but does not support changes in color for different states.

  A value display supports a change in style based on the state. For example, you can set red color for a specific threshold value. If the value in the display widget reaches this specified threshold value, the color of the widget changes to red.
- Style theme is a set of properties such as text, colors, and lines that you apply to elements in a Mashup. When you apply a style theme to a top-level Mashup, it is also applied to all embedded widgets and Mashups. You can manage styles for multiple Mashups with style themes.

Note

New widgets available in ThingWorx 8.4 and later releases, will use style themes. The style definitions will be deprecated in a future release. It is recommended to use style theme features instead.

Masters

A Master Mashup provides consistent framing for the contents of a Mashup. This is similar to a master page or template. A Master is commonly used for items that display throughout the Mashup, such as logos, menus, titles, headers, footers, sidebars, and so on. You can reuse a Master Mashup to get the same look and feel across the application.

You can create different Masters for different projects, or one for all your Mashups, depending on requirements.

Types of Mashups

You can create different types of Mashups:

- Static—Statically sized to the dimensions that you define. When displayed in a lower resolution, it has scroll bars, and in a higher resolution it leaves an unused space around the Mashup.
- Responsive—Fills to the resolution of the display. It does not leave any unused space around the Mashup.
- Responsive (Advanced)—Fills to the resolution of the display. Enables you to arrange items within a container using the advanced Flexbox-based containers.

Considerations for Developing a User Interface

Use the following best practices while developing a user interface.

Create Mockup Designs Before Creating Mashups

Before you start designing your application and creating the Mashup, it is recommended to do the following:

- Gather information from users about the features they want in the application.
- Consolidate data on the required features.
- Get information about devices that will send data to your application.
- Work with the user experience or user interface designers to develop a standard application layout. If you get the layout approved by the required stakeholders, you can then directly build the Mashup with fewer iterations.
- Determine the primary viewing device for the application, whether it will be a phone, tablet, or desktop.
- Use a responsive Mashup if you have not decided on the primary viewing device. After you select the layout type, it cannot be changed.

Use Contained Mashups to Embed Reusable User Interface

It is recommended to split your entire application into smaller reusable components. The reusable Mashups can be configured on the main Mashup page. For example, the search results page can be a reusable component. Depending on the search criteria, you can configure the Mashup accordingly.

To embed the reusable Mashup in your application Mashup, use the Contained Mashup widget. The advantages of using reusable Mashups are:

- Splitting up the Mashups enables multiple developers to work on the user interface simultaneously.
- Reusable components make it easier to develop the user interface by reducing the number of the widgets required on the application screen. With fewer widgets, you need not work with a lot of widget parameters, their bindings, and so on.
- Reusing the Mashups reduces the overall development effort.

Make the User Interface Intuitive

Strive to create intuitive interfaces to reduce the learning curve for your users. Consider the following recommendations:

- Provide a logical flow.
- Consider how your users read on their screen and design the workflows accordingly. For example, left-to-right and top-to-bottom. The layout should guide the user to perform the next action.
- Provide feedback to the user. For example:
  - Display messages that help the user understand what information to fill in or select in the user interface.
  - Provide the status of the action that the user performed.
  - If your application is executing some code in the background due to which the screen may become unresponsive, alert the user with a message. This ensures that users do not perform additional actions and wait for the application to complete the process.

Reduce the Interaction Overload

Simplify your designs to make them easy to use. Consider the following recommendations:

- Follow the minimalist UI design principles.
- Group elements with similar purpose together.
- Show only those elements that require user attention and interaction.
- If you want to show additional data, use a popup or other drill-down features. Users can access the data, as required.

- Minimize the number of clicks to navigate from one application screen to another. Use tabs to organize information on the screen and to navigate between data sets.
- Use the Enter key to trigger an action after the inputs have been entered. For example, in a text box after you type some text, you should be able to press the Enter key to submit the text.

Fewer Client Server Interactions

It is recommended to avoid unnecessary interactions between servers and clients in the application.

For example, consider a case where four services are set up to:

- Get utilization records for a specific machine
- Reverse sort the records for a different display widget
- Aggregate the records by machine state
- Calculate the utilization

If you use an Infotable, you can make a single call to the server and return all four result sets of data to the Mashup. The four services are still used; however, the services are executed in a single call.

It is recommended to use `GetProperties` service instead of defining new services (the four services mentioned in the example) to get the data. The new services pull the data from the server, even though no data may have changed. The `GetProperties` service pushes data only when the data has changed. This reduces the extra calls made to the server and eliminates unnecessary server and client interactions.

See the section Services on page 25, for more information on using `GetProperties` service.

Best Practices for Creating Mashups and Masters

Use the following best practices while creating Mashups and Masters.

Use Master Mashups for Consistency in the User Interface Design

Start by determining the basic look and feel for your application and then create a Master Mashup. Use Master Mashups to standardize the layout and display of items, such as logos, menus, titles, headers, footers, and sidebars for your Mashups.

Design Separate Mashup and Master Pages

Design separate Mashup and Master pages for different viewing devices. Remember that the user workflows are different depending on the screen sizes.

Define the Parameters for a Mashup

You can define any number of parameters for the Mashup. These parameters are used to pass data when a Mashup is embedded in another Mashup, or when navigating from one Mashup to another. These parameters become properties that can be used as binding sources or targets when they are called from other Mashups.

When a Mashup is loaded, and user did not pass all the data required by the Mashup, services are called to get this data. To minimize the number of services being called, it is recommended that the Mashups calling these parameters should pass all the data required by the containing Mashup.

Display the Sequential Tasks to Users

If you want the user to complete tasks in a sequence or have a workflow that should be performed in a certain order, use one of the following approaches:

- Set the property called `Visibile` for widgets to control the visibility of widgets. This property displays and enables widgets for required steps.
- Use dynamic Contained Mashups that display and remove the embedded Mashups after the user completes the steps in the required sequence.

Customize the Mashup to Display Selected Properties

The Property Display widget enables you to visualize a Thing property set based on their data type. A Thing property set contains all the properties and the Property Display widget displays the property set.

You can configure the Property Display widget to display only certain properties of the Mashup.

Additionally, individual properties that are retrieved from the same `getPropertyValues` service can be bound to the Value Display and other visualization widgets.

Event Log When the Mashup Loads

When you view a Mashup, you can add `&__trace` to the URL to see the detailed log of all the events that are executed when the Mashup loads. This also helps during troubleshooting.

Add `&__trace` to the end of the Mashup URL as shown:

```
<Mashup_URL>&__trace
```

Refresh Data on Mashups

You can automatically refresh data on Mashups. Use the `GetProperties` service.

Use Configuration Tables to Customize Mashups

You can use configuration tables to customize components in a Mashup. In the configuration table, provide a default value that points to a default entity such as media entity in ThingWorx.

Image Widget with Logo

You can specify a default logo for a Mashup using the image widget and a configuration value in the configuration table. Users can specify their own logo by changing the configuration value.

Contained Mashup

You can specify a default Contained Mashup in the configuration table. Users can specify their own Contained Mashup by changing the configuration value.

The out-of-the-box Contained Mashup is non-editable. To configure the Contained Mashup:

1. Create a duplicate copy of the component.
2. Customize the copy of Contained Mashup.
3. Use configuration tables or parameters to replace the original Contained Mashup with the customized Contained Mashup.

Additional Resources for Mashups and Masters

This section lists the additional resources that you can reference for a better understanding of the concepts.

Additional Resources for More In-Depth Learning

For more in-depth information on	Link	Resource
Mashups	http://support.ptc.com/help/thingworx_hc/thingworx_8_hc/en/index.html#page/ThingWorx/Help/Composer/Visualization/Mashups.html#wwID0EOHWV	Help center
Masters	http://support.ptc.com/help/thingworx_hc/thingworx_8_hc/en/index.html#page/ThingWorx/Help/Composer/Visualization/Masters/Masters.html	Help center

For more in-depth information on	Link	Resource
Mashup Builder	http://support.ptc.com/help/thingworx_hc/thingworx_8_hc/en/index.html#page/ThingWorx/Help/Mashup_Builder/MashupBuilder.html	Help center
Create Your Application UI	https://developer.thingworx.com/resources/guides/mashup-builder-quickstart	Developer portal
Effective UI Implementation	https://developer.thingworx.com/resources/guides/mashup-widget-breakdown	Developer portal

Widgets in ThingWorx Applications

Widgets are the visual components of ThingWorx Mashup pages. They include components such as dashboard widget that enables you to display other widgets, for example, a list widget.

Widgets can be used to render data or perform an action. For example, data rendering widgets, such as a chart or table, have incoming data bindings. When a data service is executed, the data result set is bound to the widget for rendering.

Widgets such as buttons may not have incoming data bindings but may have an outgoing event binding. For example, if the user clicks a button, it can trigger an event that may be outward bound to a data service. On a button click, the data service is executed on the server.

For creating your application, you can use the out-of-the-box widgets or create your own custom widgets.

Types of Widgets

In the ThingWorx Platform, the widgets are classified into three major categories:

- Standard—These are the existing widgets in the ThingWorx Platform. You can add these widgets to new and existing Mashups.
- Themeable—Widgets that you can style by applying a Style Theme.

Note

You can add these widgets only to **Responsive (Advanced)** Mashups.

- Legacy—Only the Repeater widget is listed as Legacy. This widget will be deprecated in a future release.

Depending on your application requirements, use the out-of-the-box widgets or create your own custom widgets.

See the ThingWorx Help Center for details on how to use the out-of-the-box widgets.

To create your own custom widgets, see the section Adding Custom Widgets on page 44.

When you add custom widgets, ensure that you create secure widgets. See the section Creating Secure Widgets on page 42 for more information.

Creating Secure ThingWorx Widgets

When you create customized widgets, ensure that you create secure widgets. You must account for cross-site scripting (XSS) and cross-site request forgery (CSRF) vulnerabilities. See the section on CSRF in the ThingWorx Help Center for more details.

The next sections provide best practices for creating secure widgets.

Use Encoding for HTML Output Value in Widgets

If you use widgets to display values in the user interface, it is recommended to use the HTML encoding. See the following sample code:

```
this.updateProperty = function (updatePropertyInfo) {
    if (updatePropertyInfo.TargetProperty === 'Text') {
        var labelText = Encoder.htmlEncode(updatePropertyInfo.
SinglePropertyValue);
        thisWidget.setProperty('Text', labelText);
        showLabelText(thisWidget.getProperty('Text'));
    }
}
```

In this example:

- The function `updatePropery()` is called every time the bound value for a widget property changes.
- The value from the widget is HTML-encoded using the encoder library.
- After encoding, use the Get and Set methods to access this new value.

- This step ensures that the Get methods perform extra sanitization on the values being returned to the user interface.

Use Decoding and jQuery for HTML Output Values in Widgets

If you use widgets to display values in the user interface, and have used the HTML encoding to encode values, use the HTML decoding to convert the encoded value to its original value. See the following sample code:

```
function showLabelText(labelText) {
     labelText = Encoder.htmlDecode(labelText);
     var spanText = thisWidget.jqElement.find('.label-text');
     spanText.text(labelText);
}
```

In this example:

- The function `showLabelText` is used to get the DOM element of the widget and update the value.
- Decode the value using the encoder library.
- Use the `text()` method with `jQuery` on the DOM element to display the value. This method has built-in escaping function that helps in securing the application against the XSS vulnerability.

Use Encoding for HTML Output in JavaServer Pages (JSP)

If you use JSP to display values in the user interface, it is recommended to use the HTML encoding. You can use ESAPI Utilities from OWASP to handle encoding on a contextual basis. See the following sample code:

```
<div class="org-info">
    <h1 class="orgName">
        <%=ESAPIUtilities.getEncoder().encodeForHTML(orgName)%>
    </h1>
    <p class="orgDescription"><%=ESAPIUtilities.getEncoder().encodeForHTML
(loginPrompt)%></p>
</div>
```

In this example:

- Users can create their own login pages for ThingWorx.
- The following fields require a user input for a login screen:
    - `orgName`
    - `loginPrompt`
- The data in these fields is untrusted data. There is a potential threat that the user may enter malicious characters for these fields. Therefore, it is required to run this data through the ESAPI utilities for HTML encoding.

Secure Custom User Interfaces

You can build your own custom user interfaces by developing extensions, or have your own hosted Web applications that consume the ThingWorx REST APIs. In these scenarios, you should ensure that the data is sanitized and encoded before it is displayed in the browser.

Make sure you understand the frameworks that you are implementing, and use the best practices for that environment to safeguard the application against the XSS vulnerabilities.

In the following example, an AngularJS application uses a template that was imported as an extension. You can use ThingWorx Utilities to build this workflow.

```
<div class="node-label-wrap" ng-class="{'graph-node-nonadmin': $parent.$parent.
isNonAdminUser}">
 <textarea type="text" ng-model="node.label" ng-change="$parent.$parent.
setBPUpdatesMade()">
  </textarea>
 <div class="node-label">{{node.label}}</div>
</div>
```

In this example:

- `node.label` is the untrusted user input. It is an arbitrary string. Users provide the name of the label for the workflow node.
- In the templates of AngularJS applications, if you wrap an untrusted input in double curly brace notation `{{}}`, for example `{{node.label}}`, it forces a string interpolation. The value is treated as a string and is not executed in the context of a script.

HTML Responses

PTC documents its HTML responses in a common class that filters any output through the ESAPI Utilities.

```
public void writeTableCell(String value, String id, String sClass, String sStyle)
throws Exception {
    startTableCell(id, sClass, sStyle);
    writeValue(value);
    endTableCell();
}

public void writeValue(String value) throws Exception {
    write(ESAPIUtilities.getEncoder().encodeForHTML(value));
}
```

Creating Customized ThingWorx Widgets

This section provides information on how to create customized widgets. It explains how the ThingWorx Mashup Builder and runtime interact with widgets. It provides details about the functions and APIs available for widgets.

The widgets provided by PTC are secure. Make sure the widgets you create are secure. See the section Creating Secure Widgets on page 42, for more information.

Custom widgets are added in the `/ui` folder of an extension. An extension has a defined folder structure. See the section Structure of an Extension on page 88 for more details.

The `/ui` folder contains the files required to define custom widgets. Every widget should be placed in its own subfolder in the `/ui` folder. The following files are required 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`

  The icon used for the widget in the **Widgets** tab in the Mashup Builder. The size of the icon should be 16 by 16 pixels.
- `<widgetname>.runtime.css`

  The style sheet file that defines the look and feel of the widget when you view it in the Mashup.
- `<widgetname>.runtime.js`

  The JavaScript file that defines the widget and its behavior when you view it in the Mashup.

Tip

If you are using the Eclipse Plugin, the **New Widget** action generates the source files and automatically updates the `metadata.xml` file.

Referencing Third-Party JavaScript Libraries and Files in Customized Widgets

If the custom widget uses third-party JavaScript libraries, images, and other Web artifacts, the best practice is to create a subfolder in the widget folder `/ui/<widgetname>`. Add these files in the subfolder. For example, if you create a subfolder `jslibrary`, the third-party files should be added in the following location:

`/ui/<widgetname>/<jslibrary>/`

These files can be referenced from the `*.ide.js` and `*.runtime.js` files using the following relative path:
`../Common/extensions/<extensionname>/ui/<widgetname>/<jslibrary>/`

The following example shows how to include a third-party JavaScript library and CSS into your widget code:

```
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" />');
}
```

Example of Creating a Customized ThingWorx Widget

This section describes how to create a customized widget and handle it at runtime.

Mashup Builder Code

The `<widgetname>.ide.js` file must implement widget functions to work with Mashup Builder. See the section Widget API: Mashup Builder on page 48, for more information on the functions available for widgets. Widgets can declare widget properties, services, and events in the functions.

The following sample code shows how to create a widget. The name of the widget is *MyWidget*, which has a bindable string property called *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, the widget methods must to do the following:

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

The following sample code shows a `<widgetname>.runtime.js` file for the customized widget created in Mashup Builder Code on page 46.

```
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 Available in Widgets

You can add the following features in your widgets:

- Services can be bound to events. For example:
  - A button is clicked.
  - Selected rows have changed.
  - A service is completed.
- Events can be bound to various services. For example, when you invoke a service, it navigates to a Mashup.
- Widget properties can have:
  - Data bound to it (inbound). The inbound data is specified as `"isBindingTarget": true` in the widget definition code.
  - Data that is sent back to ThingWorx (outbound). The outbound data is specified as `"isbindingSource": true` in the widget definition code.

  For example, `user_input_for_name` in the following sample code is a user input, which is an outbound data.

  ```
  this.widgetProperties = function () {
    return {
      "name": "user_input_for_name",
      "description": "Sample widget that returns a user input",
      "category": ['Common'],
      "properties": {
        "DisplayText": {
           "baseType": "STRING",
           "defaultValue": "My First Widget!",
           "isBindingTarget": true,
           "isbindingSource": true
                       }
                  }
           };
   };
  ```

You can access all the functionality of JavaScript and HTML in your widget code at runtime.

Functions Available for Customizing ThingWorx Widgets

This section describes the functions available for widgets.

Widget Lifecycle in the Mashup Builder

A widget has the following lifecycle in the Mashup Builder. In each lifecycle state, the Mashup Builder calls the following functions that are available for widgets:

- Discovered

  The widget is loaded in `index.html` and added to the widget toolbar or palette.

Function	Description
`widgetProperties()`	Get information about every widget in the Mashup. Information such as display name and description is retrieved.
`widgetEvents()`	Get information about the events that are exposed by every widget.
`widgetServices()`	Get information about the services that are exposed by every widget.

- Created

  The widget is added on the Mashup panel.

Function	Description
`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.

Function	Description
`renderHtml()`	Get the HTML fragment that is inserted in the Mashup DOM element.
`afterRender()`	Called after the HTML fragment representing the widget is inserted in the Mashup DOM element and a usable element ID is assigned to the DOM element holding the widget content. After this, the DOM element is ready to be manipulated.

- Updated

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

Function	Description
`beforeSetProperty()`	Called before a widget property is updated.
`afterSetProperty()`	Called after a widget property is updated.

- Destroyed

  The widget is deleted from the Mashup.

Function	Description
`beforeDestroy()`	Called before the DOM element of the widget is removed, and the widget is detached from its parent widget and deallocated. You should free the resources such as plugins and event handlers that are acquired during the lifetime of the widget.

Property Bag

Property bag is a set of name-value pairs. Each pair is a simple value store. ThingWorx uses property bags as a structure persistence mechanism for Mashups.

Mashup Builder APIs Available for Widgets

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 the widget has been rendered on the HTML page.
- `this.jqElement`—This is the jQuery element.
- `this.getProperty(name)`—Retrieves the name of the property.
- `this.setProperty(name,value)`—Every call to this function calls the function `afterSetProperty()`, if `afterSetProperty()` is defined in the widget.
- `this.updatedProperties()`—This function should be called when properties are changed in the widget. After the function is called, the Mashup Builder updates the widget properties window.
- `this.getInfotableMetadataForProperty(propertyName)`—Gets the InfoTable metadata for a property that is bound. It returns undefined if the property is not bound.
- `this.resetPropertyToDefaultValue(propertyName)`—Resets the specified property to its default value.

- `this.removeBindingsFromPropertyAsTarget(propertyName)`—Removes target data bindings from the specified property.
- `this.removeBindingsFromPropertyAsSource(propertyName)`—Removes source data bindings from the specified property.
- `this.isPropertyBoundAsTarget(propertyName)`—Checks if the property is bound as a target. You can use the function to determine if a property has been set or bound. For example, the following sample shows how to use `validate()` function in the blog widget to determine if a property is bound:

  ```
  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)`—Checks if the property has been bound as a source. You can use the function to determine if a property has been bound to a target. For example, the following sample shows how to use the `validate()` function in the checkbox widget to determine if a property is bound:

  ```
  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 modify the behavior of a widget.

- `widgetProperties()` [required]—Returns a JSON structure that defines the properties of the widget.

  Required properties are:

  - *name*—Name of the widget that is displayed in the widget toolbar.

  Optional properties are:

Property	Description
description	Description of the widget, which is used as its tooltip.
iconImage	File name of the widget icon or image.
category	Array of strings that specify the categories under which the widget is available. For example, **Common**, **Charts**, **Data**, **Containers**, and so on. This enables the user to filter the widgets by the category type.
isResizable	Specifies if the widget can be resized.
defaultBindingTargetProperty	Name of the property that should be used as the data or event binding target.
borderWidth	If your widget has a border, set this property to the width of the border. The property helps you ensure a pixel-perfect design during the development and at runtime. For example, consider a widget whose size at runtime should be 10X10 pixels with a border size of 1 pixel. At the design time, the size of the widget should be 8X8 pixels to handle the 1 pixel added at design time. This places the border inside the widget and makes the width and height in the widget properties accurate.

Property	**Description**
isContainer	Specifies if an instance of the widget can be a container for other widget instances. The valid values are `true` and `false`. The default value of the property is `false`.
customEditor	Name of the custom editor dialog that should be used to set and edit the widget configuration. In your widget project create a dialog box with the name `TW.IDE.Dialogs.<name>`.
customEditorMenuText	The text that appears on the flyout menu of the widget. It is also the text that appears as the tooltip for widgets in the ThingWorx Composer . For example, **Configure Grid Columns**.
allowPositioning	Checks if the widget can be placed on a panel. The valid values are `true` and `false`. The default value of the property is `true`.
supportsLabel	Specifies if the widget supports the Label property. The valid values are `true` and `false`. The default value of the property is `false`. When set to `true`, the widget exposes a Label property. This property is used to create a text label that appears next to the widget in the ThingWorx Composer and at runtime.

Property	**Description**
supportsAutoResize	Specifies if the widget automatically supports resizing. The valid values are `true` and `false`. The default value of the property is `false`. When set to `true`, the widget can be placed in responsive containers, such as, columns, rows, responsive tabs, and 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. The corresponding attributes control 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:

Attribute	**Description**
description	Description of the widget used for its tooltip.
baseType	Base type of the widget. If the base type value is `FIELDNAME`, the widget property window displays a list of fields available in the

Attribute	Description

	InfoTable. The fields are bound to the `sourcePropertyName` value based on the base type restriction.
	
	The other special base types are:
	• STATEDEFINITION—Selects a state definition.
	○ STYLEDEFINITION—Selects a style definition.
	○ RENDERERWITHSTATE—Displays a dialog that enables you to select a data renderer and state formatting.
	
	Note
	You can set the default style in the *defaultValue* attribute. Specify the string with the default style name in *defaultValue*. When your binding changes, you should reset the property to the default value, as shown in the code below:
	`this.afterAddBindingSource = function (bindingInfo) {`
	`if(bindingInfo['targetProperty'] === 'Data') {`
	`this.resetPropertyToDefaultValue('ValueFormat');`
	` }`
	`};`
	○ STATEFORMATTING—Displays a dialog that enables you

Attribute	Description
	to select a fixed style or state-based style. **Note** You can set the default style in the *defaultValue* attribute. Specify the string with the default style name in *defaultValue*. When your binding changes, you should reset the property to the default value. See the sample code provided in RENDERERWITHSTATE. ○ VOCABULARYNAME—Selects the data tags vocabulary at runtime.

mustImplement	If the base type is THINGNAME and you specify the *mustImplement* attribute, the Mashup Builder restricts the dialog box to open only those popups that implement the specified EntityType and EntityName. The Mashup builder calls the function `QueryImplementingThings` to validate if the value passed is of the type EntityType and checks if EntityName is implemented.
	
	For example:
	`'baseType': 'THINGNAME',`
	`'mustImplement': {`
	` 'EntityType': 'ThingShapes',`
	` 'EntityName': 'Blog'`
	`}`

baseTypeInfotableProperty	If the base type of the widget is set as RENDERERWITHFORMAT, the attribute specifies which property of the InfoTable is used for configuration.
sourcePropertyName	If the base type of the widget is set as FIELDNAME, the attribute checks

Attribute	Description
	which property of the InfoTable is used for configuration.
baseTypeRestriction	If this attribute is specified, the value is used to restrict the fields available in the FIELDNAME drop-down list.
tagType	If the base type is set as TAGS, the attribute specifies the tag type. The valid values are `DataTags`, which is the default value or `ModelTags`.
defaultValue	Used to set the default value for a property. The default value is undefined.
isBindingSource	Specifies if the property is a data binding source. The valid values are `true` and `false`. The default value of the property is `fasle`.
isBindingTarget	Specifies if the property is a data binding target. The valid values are `true` and `false`. The default value of the property is `fasle`.
isEditable	Specifies if the property is editable in ThingWorx Composer. The valid values are `true`, which is the default value and `false`.
isVisible	Specifies if the property is visible in the properties window. The valid values are `true` and `false`. The default value of the property is `true`.
isLocalizable	Specifies if the property can be localized. This attribute is required when the base type is set to STRING. The valid values are `true` and `false`.

selectOptions	An array of value or display text structures.
	
	For example:
	`[{value: ‘optionValue1’, text: ‘optionText1’},`
	`{value: ‘optionValue2’, text: ‘optionText2’}]`

Attribute	Description
warnIfNotBoundAsSource	ThingWorx Composer checks if the property is a data binding source. The valid values are `true` and `false`. If the attribute value is set to `true`, but the property is not bound, ThingWorx Composer generates a list of items that you must complete to save the Mashup.
warnIfNotBoundAsTarget	The composer checks if the property is a data binding target. The valid values are `true` and `false`. If the attribute value is set to `true`, but the property is not bound, ThingWorx Composer generates a list of items that you must complete to save the Mashup.

- `afterLoad()` [optional]—The function is called after your object is loaded and properties are restored from the file, but before your object is rendered on the HTML page.
- `renderHtml()` [required]—The function returns the HTML fragment that ThingWorx Composer places on the screen. The content container of the widget, such as, `div`, should have a widget-content class specified in it. After this, the container element is appended to the DOM. The container is accessible using the `jqElement`, and its DOM element ID is available in `jqElementId`.
- `widgetEvents()` [optional]—A collection of events; each event can have the following property:
    - `warnIfNotBound`—The composer checks if the property is bound. The valid values are `true` and `false`.

      If the attribute value is set to `true`, but the property is not bound, ThingWorx Composer generates a list of items that you must complete to save the Mashup.
- `widgetServices()` [optional]—A collection of services; each service can have the following property:
    - `warnIfNotBound`—The composer checks if the property is bound. The valid values are `true` and `false`.

If the attribute value is set to `true`, but the property is not bound, ThingWorx Composer generates a list of items that you must complete to save the Mashup.

- `afterRender()` [optional]—The function is called after the HTML fragment is inserted in the DOM.
- `beforeDestroy()` [optional]—The function is called before the DOM element of the widget is removed and the widget is detached from its parent widget and dellocated. You should free the resources such as plugins, event handlers, and so on, acquired throughout the lifetime of the widget.
- `beforeSetProperty(name,value)` [optional] [Mashup Builder only - not at runtime]—The function is called before any property is updated in the ThingWorx Composer. You can perform validations on the new property value before it is committed. If the validation fails, you can return a message string to inform the user about the invalid input. The new property value is not be committed. If nothing is returned during the validation, then the value is assumed valid.
- `afterSetProperty(name,value)` [optional] [Mashup Builder only - not at runtime]—The function is called after any property is updated in the ThingWorx Composer. If you specify the value as `true`, the widget is rendered again in the ThingWorx Composer after the value is changed.
- `afterAddBindingSource(bindingInfo)` [optional]—This function is called whenever data is bound to your widget. The only field in `bindingInfo` is `targetProperty`, which is the name of the property and is bound.
- `validate()` [optional]—The function is called when ThingWorx Composer refreshes its to-do list. The call returns an array of result objects:
  - *severity* —This is an optional property. The severity of the problem that was detected.
  - *message* —This a required property. The message text may contain one or more predefined tokens, such as `{target-id}`. The message is replaced with a hyperlink that enables the user to navigate and 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;
}
```

Runtime Functions Available for Customizing ThingWorx Widgets

This section describes the functions available for widgets at runtime.

Widget Lifecycle in Runtime

- When a widget is first created, the runtime obtains any declared properties by calling the `runtimeProperties()` function.

  See the section Callbacks from Runtime to Your Widget for more information.
- The property values that are saved in the Mashup definition are loaded into your object without calling any function.
- After your widget is loaded but before it is rendered on to the screen, the runtime calls the function `renderHtml()`. In this function, you return the HTML for your object. The runtime renders the HTML returned by the function in the appropriate place in the DOM.
- After the HTML is added to the DOM, the function `afterRender()` is called. If required, you can perform `jQuery` bindings. At this point, you can reference the actual DOM elements. For example, use the following code to reference the DOM elements:
  ```
  // note that this is a jQuery object
  var widgetElement = this.domElement;
  ```

  Note

  The runtime changes the ID of your DOM element. It is recommended to always use the `jQuery` object `this.domElementId` to get the ID of the DOM element.

- If you have defined an event that can be bound, you can trigger the event, as shown in the following example:
  ```
  var widgetElement = this.domElement;
  // change ‘Clicked’ to be your event name
  // you defined in your runtimeProperties
  widgetElement.triggerHandler('Clicked');
  ```
- If you have any properties bound as data targets, the function `updateProperty()` is called. You should update the DOM directly if the changed property affects the DOM. When the data is bound, in most cases the DOM changes.

- If you have properties that are defined as data sources and they are bound, you can call the function `getProperty_{propertyName}()`. If you do not define this function, the runtime gets 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 the widget is rendered on the HTML page.
- `this.jqElement`—This is the `jQuery` element.
- `this.getProperty(name)`—Gets the name of the property.
- `this.setProperty(name,value)`—Sets the name and value of the property.
- `this.updateSelection(propertyName,selectedRowIndices)`—Call this function when your widget changes the property on the selected rows. These rows have data bound to it. For example, in a callback if you have defined an event such as `onSelectStateChanged()`, then call the API `this.updateSelection(propertyName,selectedRowIndices)` and the system updates all the widgets that are dependent on the 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 the widget.

  Optional properties are:

Property	Description
isContainer	Specifies if an instance of the widget can be a container for other widget instances. The valid values are `true` and `false`. The default value of the property is `fasle`.
needsDataLoadingAndError	The valid values are `true` and `false`. The default value of the property is `fasle`. Set the property to `true` if you want the widget to display the standard 25% opacity when no data is received. The widget turns red when there is an error while

Property	**Description**
	retrieving the data.
borderWidth	If widget has a border, set this property to the width of the border. The property helps you ensure a pixel-perfect design during development and runtime.
supportsAutoResize	Specifies if the widget automatically supports resizing. The valid values are `true` and `false`.

propertyAttributes	If you have STRING properties that are localizable, list them in this property.
	
	For example, if `TooltipLabel1` is localizable:
	`this.runtimeProperties =`
	`function () {`
	`return {`
	` 'needsDataLoadingAndError':`
	`true,`
	` 'propertyAttributes': {`
	`  'TooltipLabel1':`
	`{'isLocalizable': true} }`
	`  }`
	`};`

- `renderHtml()` [required]—The function returns the HTML fragment that the runtime places on the screen. The content container of the widget, for example, `div` should have a widget-content class specified in it. After the class is specified, the container element is appended to the DOM. The container is accessible using `jqElement` and its DOM element ID is available in `jqElementId`.
- `afterRender()` [optional]—This function is called after the widget HTML fragment is inserted in the DOM. Use `this.domElementId` to get the DOM element ID. Use `this.jqElement` element to get the jQuery DOM element.
- `beforeDestroy()` [optional but highly recommended]—Call this function when the widget is removed from the screen. With this function, you can:
  - Unbind any bindings
  - Clear any data set with `.data()`
  - Destroy any third-party libraries or plugins, call their destructors, and so on.

- Free any memory you allocated or are holding in closures, by setting all your variables to `null`
- You do not need to destroy the DOM elements inside the widget, they are destroyed by the runtime

- `resize(width,height)` [optional – Useful only if you declare supportsAutoResize: true]—This function is called when your widget is resized.

  Some widgets do not need to handle this. For example, if the elements of the widget and CSS auto-scale.

  However, most widgets require handling when the widget size changes.

- `handleSelectionUpdate(propertyName, selectedRows, selectedRowIndices)`—This function is called when the selected rows are modified by the data sources bound to it with the specified property. `selectedRows` is an array of the actual data, and `selectedRowIndices` is an array of the indices of the selected rows.

Note

You must define this functionality to get the full `selectedRows` event functionality without having to bind a list or grid widget.

- `serviceInvoked(serviceName)`—The function is called when 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, the `updateProperty()` of your widget is called every time the source data is changed. You should check `updatePropertyInfo.TargetProperty` to determine what aspect of your widget should be updated.

See the following example for a widget:

```
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);
   }
 }
};
```

Set a local copy of the property in the widget object. This ensures that the runtime system can get the property from the property bag.

Alternatively, you can provide a custom `getProperty_{propertyName}` method and store the value in some other way.

- `getProperty_{propertyName}()`—When the runtime requires the 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 getting the data from your widget to populate parameters for a service call.

Tips for Creating Customized ThingWorx Widgets

This section provides tips for creating customized widgets.

- Use `this.jqElement` to limit your element selections. This reduces the chance of introducing unpredictable behavior in the application if there are duplicate IDs and classes in the DOM.

  Use `this.jqElement`, as shown in the sample code to limit your element selections:
  ```
  this.jqElement.find('.add-btn').click(function(e){
  ...do something...});
  ```
- Logging—You can create logging events for debugging Mashups. It is recommended that you use the following methods to create log messages in the 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])`

  Logs are available under the **Monitoring** menu. Select a log to open the log window to view the log messages. If the browser you use supports `console.log()`, then the messages also appear in the debugger console.
- Formatting—If you have a property with base type set as STYLEDEFINITION, you can get the style information using the following code:
  ```
  var formatResult = TW.getStyleFromStyleDefinition(
   widgetProperties['PropertyName']);
  ```
  If you have a property of base type set as STATEFORMATTING, use the following code:
  ```
  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'
};
```

Best Practices for Creating Mashups Using Widgets

Use the following best practices while creating Mashups with widgets.

Add Layout as Top-Level Widget in a Mashup

It is recommended that you add a Layout widget at the top-level of any container even if you add one widget. Adding a top-level Layout widget gives you the flexibility to add an additional row or column to the area as well as a header or footer in future.

Visibility of Widgets on a Mashup

You can manipulate the visibility of ThingWorx widget at runtime using the widget property Visible.

Validating User Input

Use the Validator widget to validate the input that the users want to send to a device. These inputs may change the state of the device, operate the device remotely, and so on. It is recommended to check if the user input is valid. You can use one Validator widget for each field you want to validate, or one for more fields if they require the same validations.

Validators accept regular expressions (regex) as input, which enables you to check almost any type of content. If the validation returns false, you can display a status message to the user, and reset the input field to the default value.

Mathematical Conversions

Use the Expression widget to perform mathematical conversions at runtime. For example, consider a case where the data about a temperature is displayed in degree Celsius (C). You can provide a radio button that enables a user to see the temperature details in Fahrenheit (F). Based on the user selection of the unit of measurement, the Expression widget calculates the value of the temperature and displays it.

It is recommended to use the Expression widget rather than a server-side custom service for simple calculations.

Additional Resources for Widgets

This section lists the additional resources that you can reference for a better understanding of the concepts.

Additional Resources for More In-Depth Learning

For more in-depth information on	Link	Resource
Mashup Builder	http://support.ptc.com/help/thingworx_hc/thingworx_8_hc/en/index.html#page/ThingWorx/Help/Mashup_Builder/MashupBuilder.html	Help center
Widgets	http://support.ptc.com/help/thingworx_hc/thingworx_8_hc/en/index.html#page/ThingWorx/Help/Mashup_Builder/Widgets/Widgets.html	Help center
Cross-site request forgery (CSRF)	http://support.ptc.com/help/thingworx_hc/thingworx_8_hc/en/index.html#page/ThingWorx/Help/REST_API/UpdatingtheRequestMethodandContentTypeFilteringforCSRFProtection.html	Help center
Basic Mashup Widgets	https://developer.thingworx.com/resources/guides/basic-mashup-widgets	Developer portal
Bind Data to Widgets	https://developer.thingworx.com/resources/guides/data-widgets-and-building-experience	Developer portal

Creating Localization Tables

Localization enables you to display runtime labels in different languages or terminology. In ThingWorx, you can configure localization tables with localization tokens. These localization tokens are assigned to text in Mashup Builder. You can use localization tokens to do the following tasks:

- Apply localization tokens to dates for locale-specific formatting.
- Translate the labels in your extension.

Best Practices for Creating Localization Tables

Use the following best practices while creating Localization Tables:

- If your extension would be localized, create localization tables while developing your user interface. When you need to present localized label and message (non-data text) to your users, use a localization token and create new tokens as needed. It is much easier to do this during initial development than finding and replacing 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 is converted to an underscore. For example, if `aspects.friendlyName = "myNamespace.myKey"`, the localization token lookup is `myNamespace_myKey`.
- To avoid conflicts with system tokens or tokens from other extensions, use a prefix or suffix that is specific to your extension. For example, instead of `NoNameProvided`, use *`MyExtension`*`.NoNameProvided`.
- Export localization tables using the token prefix filter set to the correct namespace, and include the exported localization tables in the extension package.

Additional Resources for Localization Tables

This section lists the additional resources that you can reference for a better understanding of the concepts.

Additional Resources for More In-Depth Learning

For more in-depth information on	Link	Resource
Localization Tables	http://support.ptc.com/help/thingworx_hc/thingworx_8_hc/en/#page/ThingWorx%2FHelp%2FComposer%2FSystem%2FLocalizationTables%2FLocalizationTables.html%23	Help Center
Packaging Extensions with Localization Tables	http://support.ptc.com/help/thingworx_hc/thingworx_8_hc/en/#page/ThingWorx%2FHelp%2FComposer%2FSystem%2FLocalizationTables%2FPackagingExtensions-withLocalizationTables.html%23	Help Center

4

Securing the Applications Built on the ThingWorx Platform Using Visibility and Permissions

Security is a critical component for all applications. You need visibility and control over your IoT solution.

ThingWorx offers a comprehensive set of tools and features that enable you to efficiently manage your connected systems, applications, and devices.

When you define the security requirements for your application, think about the following questions:

- Who will use the application? Will it be used by a single customer (company) or will it be shared by multiple customers?
- What are the different types of users for your application? What visibility and permissions should the user types have in the application?
- Will the user interface change, based on the User type? For example, some users will need full access to all components of the user interface, while other users will have access to only certain Mashups or menu items.

Configuring Visibility and Permissions for ThingWorx Entities

In ThingWorx, there are two types of permissions:

- Design time—These permissions define which User Groups and Users have access to create, read, update, and delete entities.
- Runtime—These permissions define which User Groups and Users can access data, execute services, and trigger events on a Thing, which include Data Tables, Streams, and Users. You can set runtime permissions at a Thing, Thing Template, or entities collection level. The abstract entities, Thing Shapes, Data Shapes, and User Groups in the model do not have runtime permissions.

Visibility permissions define which Organization and organizational units have access to view an entity. If the members of an Organization or organizational units do not have visibility access, they are not able to view the entity in the entity list or search results.

It is recommended that you manage the permissions for your application as you build it. Implementing permissions at the end is very difficult.

Note that by default security checks do not allow an operation. If no specific permission is given to a user, then that operation is denied.

Avoid assigning unnecessary permissions to a group, as it might expose your application to unwarranted risks. You can use the override functionality to deny permissions on services that are available on the platform.

Recommended Workflow While Defining Visibility and Permissions for Entities

When you define visibility and permissions for entities such as Things, Mashups, and so on, the recommended workflow is:

1. Define Organizations and organizational units based on your company structure.
2. Create User Groups and add them to the Organizations or organizational units.
3. Create Users and add them to User Groups.
4. Assign visibility for entities based on Organization or organizational units.
5. Assign permissions for entities based on User Groups or Users.

Use the following best practices while defining visibility and permissions.

Runtime Instance Permission for User Groups for All Things

Use runtime instance permission to assign permissions to User Groups for all Things that implement a specific Thing Template, rather than assigning permission to each Thing individually.

Import User Groups and Users Before Other Types of Entities

Collections may not be written as expected to the application if the corresponding User Groups or Users do not exist before you import **From Thingworx Storage** on the target server. To ensure that the permissions import properly, create the User Groups and Users in ThingWorx. Export the User Groups and Users from the system as an export file in binary or XML format. Then import them back into the same or new system, and finally import the rest of your model.

Import and Export of Collection-Level Permissions

Collection-level permissions for Things, Thing Templates, Logs, and so on, are exported or imported only when you export or import **From Thingworx Storage**. If you import or export **From File**, the collection-level permissions apply only on the entity level.

Reset the Default Administrator Password

Warning: It is critical to reset the Administrator password after your first login. If you do not change the default password, it can allow the system to be compromised in the future.

Configuring Visibility and Permissions for Organizations, User Groups, and Users in ThingWorx

Organizations and User Groups are used to provide visibility and assign permissions to Users as a group. You can also provide visibility and assign permissions to individual Users.

Permissions Assigned at User Group Level

Permissions must be assigned at the User Group level rather than at an individual User level. Assigning permissions at the User Group level makes it easy to manage the permissions in an application. The only exception is the System user, who has permissions at the User level.

Test User Group Permissions

It is recommended to test all User Groups and their permissions before deploying your application. This ensures that the User Groups have access to the right functionality based on their visibility and permissions.

User Group Services

Some services are available through a Resource called `EntityServices`. The Resource enables you to interact with User Group entities programmatically. After you create the User Group, you can interact with it through built-in services to add or remove a user.

These services allow you to write custom services that set visibility and permissions on collection of entities.

Remove Users Group from Everyone Organization

Remove the out-of-the-box Users group from the Everyone Organization. This ensures that all the Users do not have visibility access to all the entities on the platform. It is recommended to assign visibility access to Users depending on the type of User.

Best Practices for General Security of ThingWorx Applications

ThingWorx provides features such as single sign-on authentication, directory services authentication, creation of authenticators and application keys to manage security of your applications. See the ThingWorx Help Center for more information.

Application Key for Communication

It is recommended that you authenticate the data sent by a connected device to the ThingWorx Platform using application keys for such authentication.

The application key is associated with a user. Users represent an individual person or connected system. The key has all permissions that are granted to the user. It is recommended that you use the principle of the least privilege while creating and assigning privileges to application keys.

It is not recommended to assign a member of the Administrator group to an application key. If administrative access is necessary, create and add the User as a member of the SecurityAdministrators and Administrator User Groups.

For encrypted communications, use HTTPS.

IP Whitelisting for Application Keys

It is recommended that you set the IP whitelist for the application key. This enables the server to specify that only certain IP addresses should be able to use a given key ID for access. You can specify a single IP in case of static IP address. For example, connected web-based business systems can have a static IP, from which all the calls are made. You can use wildcards to specify a range of IPs addresses for devices with dynamic IP addresses.

Whitelisting is not recommended for devices that continually change networks and IP addresses. They may lose the ability to connect when the IP whitelist feature is used.

Additional Resources for Security, Visibility and Permissions

This section lists the additional resources that you can reference for a better understanding of the concepts.

Additional Resources for More In-Depth Learning

For more in-depth information on	Link	Resource
Security	http://support.ptc.com/help/thingworx_hc/thingworx_8_hc/en/index.html#page/ThingWorx/Help/Security/Security.html	Help Center
Configure permissions	https://developer.thingworx.com/en/resources/guides/data-model-security-control	Developer portal
Thingworx Organizations and visibility	http://youtu.be/ZYvKqKY9KTk	Video

5

Building ThingWorx Applications Using Reusable Components

This section provides a brief overview on how to develop applications using components that can be reused throughout your applications. It also provides best practices for configuring and customizing these reusable components, as well as creating your own reusable components.

When you build your applications, it is efficient to reuse standard components throughout your applications. These reusable components become the building blocks of your application, and you can configure and customize these components to suit your business case.

Reusable components provide the following benefits:

- Ensure consistent design practices thought your application suite.
- Allows efficiency and flexibility when creating new or modifying existing applications to address your evolving business needs.
- Provide a better handling of the source data, as the components contain prebuilt business logic.
- Save on the design and development time.
- Ensure the use of best practices when building applications.

Many ThingWorx entities such as widgets with business logic, services, data adapters can be reused in different applications.

Terms Used in Reusable Components

The following terms are used for reusable components:

Term	Definition
Component	A container that contains a set of standard ThingWorx entities.
Atomic component	The smallest component that cannot be divided during the design time.
Composite component	A component that comprises of atomic and other composite components.
Configurable	The behavior of a component can be changed by configuring the exposed parameters.
Customizable	The out-of-the-box capability of a component can be changed by updating the internal code and entities.
Extendable	Add additional capabilities to the component by updating the properties and functionality.
Upgradable	The component supports in-place and migration upgrades.
Runtime	The execution time of the application on the ThingWorx server.
Design time	The time required to develop the application using the ThingWorx Composer and Mashup Builder.
Domain	A business unit that contains some components.
Module	A group of components under the same domain.

Types of Reusable Components

The reusable components are categorized as:

- Atomic components on page 77
- Composite components on page 79

Atomic Components

Atomic components are made up of ThingWorx modeling entities. User interface widgets are examples of atomic components.

Types of Atomic Components

The types of atomic components are:

- Model
- Services
- Data Adapter
- User interface

Creating Model Atomic Component

Consider the following best practices when you create the model atomic component:

- To create a component with configurable parameters, use configuration tables on Thing Templates. In the configuration table, you can define and collect data for parameters that should be configurable.
- Things represent physical devices, assets, products, systems, people, or processes that have properties and business logic. An application should work even if the Things just implement all the required Thing Shapes without extending the out-of-the-box Thing Templates.
- A Thing can have Data Tables and Value Streams associated to it. If the Thing along with the model entities is shared with a group of physical assets, then the Data Table and Value Stream must also be shared.
- Extend and customize the component, as required. See the section Extending and Customizing a Component on page 83, for more information.

Creating Service Atomic Component

Consider the following best practices when you create the service atomic component:

- It is recommended to use Thing Templates and Thing Shapes to create services for an atomic component. Whenever possible, define the services in Thing Shapes. See the section Creating Components Using Thing Shapes on page 83, for more details.
- If possible, implement services in JavaScript.
- Users should be able to override the services. This enables the users to replace the out-of-the-box services with their own customized services.

- It is recommended not to share the services. If you want to reuse services defined in a Thing Shape or Thing Template in an application, you must create a new Thing implementing the Thing Shape or Thing Template that contains the required service.
- Extend and customize the component as required. See the section Extending and Customizing a Component on page 83, for more information.

Creating a Data Adapter Atomic Component

The best practices recommended for services also apply to data adapters. In addition, consider the following best practices when you create the data adapter atomic component:

- Implement security, authentication, and access control in this layer.
- Minimize the business logic related to the application in this component.
- Data adapter is a pure data fetching layer. It does not implement any business logic.
- While working with large data sets, the data adapter must support pagination.
- An adapter can be shared with various components such as models and services.
- Data aggregation happens at the service component level.

Creating a User Interface Atomic Component

Consider the following best practices when you create the user interface atomic component:

- Use ThingWorx widgets to create the user interface. The widgets should not contain any ThingWorx services because the user interface should not contain any business logic.

Note

These widgets are the out-of-the-box ThingWorx widgets. They have not been created using the SDK.

- Develop the user interface using JavaScript.
- The following visualization elements should be non-editable to support upgrades:
  - Media
  - Style
  - State definition
  - Static menu

Composite Components

A composite component can contain a set of atomic components, or a set of other composite components, or a combination of atomic and composite components. The composite components have some business logic embedded in them. The ThingWorx Contained Mashup, widgets created using SDK are some examples of composite components.

Use the following best practices while creating composite components.

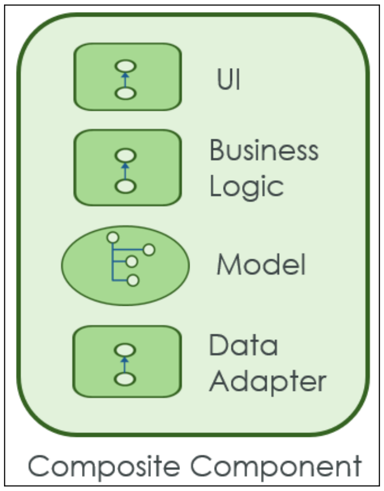
Subcomponents of Composite Components

The composite component consists of all or some of the following subcomponents:

- User interface (UI)
- Business logic
- Model along with services
- Data adapter

When you use a composite component, one of these subcomponents is considered as the top-level component. The top-level component is used as the access point for the composite component.

The composite component establishes the top-level component using the hierarchy of subcomponents. The highest in the hierarchy of subcomponents is the UI, followed by the business logic, model along with services, and data adapter as shown in the following figure:



For example, if a composite component has the UI defined for it, then the UI is the access point for the component. If the composite component has no UI and has the model defined for it, then access point is the model along with services. If no UI or model is defined for a component, then the access point is the data adapter.

It is recommended not to expose all the subcomponents below the top-level component to users.

Types of Composite Components

The following types of composite components are available:

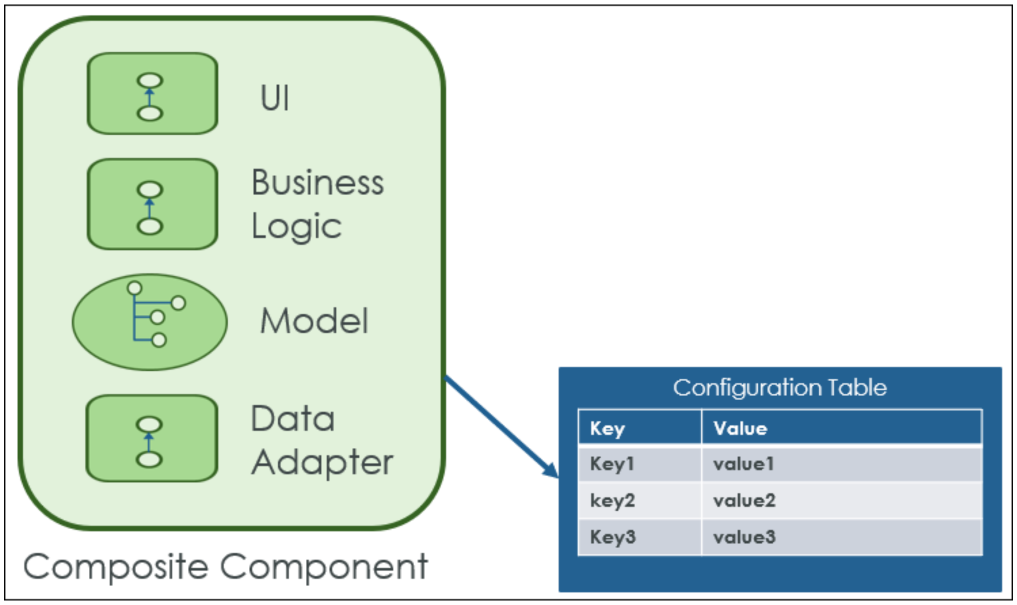
- Contained Mashup—Non-editable, can be configured using parameters or customized using custom CSS.

  It is recommended to keep the number of parameters to a minimum, whenever possible.
- Widgets created using SDK—Non-editable, can be configured using parameters or customized by inheritance.
- Business logic—Use a configuration table or parameters for customization.

  For example, a business logic which consists of the component model and data adapter.
- Gadgets—Non-editable
- Dashboard—Non-editable
- Dynamic menu

Configuring the Composite Components

You can configure composite components using a configuration table, parameters, or configuration Mashups. The configuration values are passed to the top-level component, which further passes the values to the downstream components in the dependency hierarchy. For example, you can use a configuration table with key-value pairs to change the behavior of the component. The configuration table is not a part of the component upgrade.



Key	Value
Key1	value1
key2	value2
Key3	value3

Upgrading the Composite Components

A composite component and its subcomponents should not be editable to support upgrades. When you upgrade subcomponents, the composite component should not break.

Customizing and Extending the Composite Components

Extend and customize the composite component, as required. See the section Extending and Customizing a Component on page 83, for more information.

For example, consider a case where you want to customize and extend an out-of-the-box Contained Mashup. The out-of-the-box Contained Mashup is non-editable. To configure and extend this Contained Mashup, create a copy of the component. Customize and extend the copy. Use configuration tables or parameters to replace the original Contained Mashup with the customized Contained Mashup.

Naming Guidelines for Reusable Components

Follow these general naming guidelines for reusable components:

- You can specify any namespace for your component. You can prefix your company name before the component name. For example, `<company_name>.<component_name>`.

PTC uses the `<company_name>.<component_name>` naming convention for its components. For example, the component name starts with PTC.*.

- The localization tokens that belong to the same component should contain the full component name as the prefix.
- All the entities related to the same component should have the same tag and belong to the same project. Their names should have the same prefix.
- Provide meaningful names to the entities.
- Avoid ambiguous names and long entity names.
- Try to include a good description for every entity you create.
- Rename the services or properties with conflicting names. For example, consider two Thing Shapes with services or properties with the same name. If these Thing Shapes are implemented by the same Thing Template, a conflict in the name occurs.
- You can have a centralized location to list the existing prefixes for components and their entity names. This helps in avoiding name conflicts.

See the section on naming entities in the ThingWorx Help Center for more information.

Best Practices for Creating Reusable Components

Use the following best practices while creating reusable components.

Packaging of Components

Individual components are packaged as extensions. Based on the business decision, a combination of components is further bundled as extensions.

Dependency of Components

Package the components into bundles for each release. The component dependency is managed through the bundle (extension) dependency. See the section Dependencies on page 89, for more information about extension dependencies.

Licensing

Based on the business logic the components are packaged together as extensions. These extensions are licensed.

Creating Non-Editable Components

It is recommended to create components that are non-editable. Non-editable components are easy to upgrade. However, ensure that the component can be configured. Expose parameters that will help users to customize the component to suit their requirements. Define if a parameter of the reusable component is visible to the user. You can set the default values for the parameters. Users should be able to customize the component by changing its internal code.

⚠ **Caution**

The API functions of the component should not be removed or modified to ensure error-free component upgrades.

Creating Components Using Thing Shapes

It is recommended to use Thing Shapes while creating components.

- Properties and services should be implemented on a Thing Shape and not on Thing Templates and Things, whenever possible.
- Use a Thing Template to group Thing Shapes. This supports inheritance.
- The services on Thing Shapes and Thing Templates should allow an override.
- The out-of-the-box Thing Shapes and Thing Templates should be non-editable.

Extending and Customizing a Component

Use the following best practices when you extend or customize a component:

1. Create new Thing Templates that inherit the out-of-the-box Thing Templates.
2. Override the services and properties in the new Thing Template.
3. Create a new Thing.
4. Introduce new services and properties in the new Thing.

Best Practices for Packaging and Versioning Reusable Components

Use the following best practices while packaging and versioning reusable components.

Packaging Reusable Components

Consider the following points while packaging the components:

- The version of a component should be based on the version of its extension.

- It is important to finalize the name of the extension and the location of components in an extension. Once an extension is created, it is difficult to rename it and move the components in the extension. See the section Extension Name and Version Convention on page 88 for more information.
- If two components are dependent on a common third component, ensure that the common component is packaged as a separate extension.

  For example, consider three components A, B, and, C, which are bundled as individual extensions. Based on these three components, two extensions are created with a combination of components.

  - Extension1—Contains components A and B.
  - Extension2—Contains components C and B.

  In this case, Extension1 and Extension2 both are dependent on component B. Bundling component B as separate extension ensures that the dependencies are easily resolved.

See the section Packaging and Deploying Applications Built on the ThingWorx Platform on page 86 for more information on packaging extensions.

Versioning Reusable Components

Consider the following points while versioning the components:

- Versioning is applied to extensions. To version components, you must bundle them in extensions.
- Use the `<major>.<minor>.<patch>` format to version an extension. Extensions follow the semantic versioning rules. See Semantic Versioning for more information.

  See the section Packaging and Deploying Applications Built on the ThingWorx Platform on page 86 for more information on versioning extensions.

Additional Resources for Reusable Components

This section lists the additional resources that you can reference for a better understanding of the concepts.

Additional Resources for More In-Depth Learning

For more in-depth information on	Link	Resource
Mashup Builder	http://support.ptc.com/help/thingworx_hc/thingworx_8_hc/en/index.html#page/ThingWorx/Help/Mashup_Builder/MashupBuilder.html	Help center
Gadgets	http://support.ptc.com/help/thingworx_hc/thingworx_8_hc/en/index.html#page/ThingWorx/Help/Composer/Visualization/Dashboards/Gadgets.html	Help center
Contained Mashup Widget	http://support.ptc.com/help/thingworx_hc/thingworx_8_hc/en/index.html#page/ThingWorx/Help/Mashup_Builder/Widgets/ContainedMashupWidget.html	Help center
Dashboards	http://support.ptc.com/help/thingworx_hc/thingworx_8_hc/en/index.html#page/ThingWorx/Help/Composer/Visualization/Dashboards/Dashboards.html	Help center
Data Model Introduction	https://developer.thingworx.com/en/resources/guides/data-model	Developer portal
Design Your Data Model	https://developer.thingworx.com/resources/guides/data-model-design-tutorial	Developer portal
Customizable and Responsive UI	https://developer.thingworx.com/resources/guides/customizable-responsive-ui	Developer portal

6

Packaging and Deploying Applications Built on the ThingWorx Platform

ThingWorx Platform is a complete end-to-end technology platform. You can use it to develop, deploy, and extend IoT applications.

The IoT applications can be deployed in two ways:

- As an extension—An application can be packaged as an extension in a ZIP file. An extension is a collection of entities, resources, and widgets. Users can import extensions into the ThingWorx Platform to work with the IoT solution.

  PTC recommends deploying your application as an extension. With extensions it is easier to manage the collection of ThingWorx artifacts. If you are building applications that will be distributed to your customers, it is recommended that you deploy the applications as extensions.

  When you create extensions, define your design requirements well in advance. For example, in the initial stages of development, you need to decide whether you want to use a Thing Shape or Thing Template for your modeling requirements, what Organizations and User Groups you need, what permissions to give Users, and so on.

- As a collection of entities—An application can be packaged as a collection of entities that can be exported as an XML or binary file if the application is an in-house lightweight solution.

Note

The recommended practice is to bundle the application as an extension.

What is an Extension?

An extension is a collection of entities, resources, and widgets that are used to extend the functionality of the ThingWorx Platform. This collection is packaged as a ZIP file that can be imported to any ThingWorx Platform. It is used to add new functionality.

Entities are created using the ThingWorx Composer. You can create widgets, resources, and Java code using external tools 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 an easy deployment.

Extensions can be deployed on the PTC Marketplace to make them accessible to the PTC customer community.

Why Build an Extension?

Some reasons to build an extension include:

- Your solution includes multiple entities that are functionally interdependent.
- Your solution depends on a Java library that is not accessible within the ThingWorx Platform.
- You want to hide your source code from those who use the extension.
- You want to use a custom widget that does not exist on the ThingWorx Platform.
- You want a global service that is not associated with an entity resource.
- Your organization wants to use a custom directory service or user authorization scheme.

Best Practices for Packaging and Deploying ThingWorx Applications

Use the following best practices while developing and deploying extensions.

Develop on a Clean Platform Instance

ThingWorx does not provide safeguards or sandboxing for your code. It is possible that sometimes your environment may become unstable. To get the application out of the unstable state, do the following:

- If you see errors in the development process, restart the Apache Tomcat server.
- You may be prompted to remove the `ThingworxStorage` directory completely. It is recommended that you develop the application on a clean instance of the platform to ensure that no work is lost when you remove the `ThingWorxStorage` directory.

Structure of an Extension

The artifacts of an extension must be packaged into a ZIP file in the following folder structure:

- `/metadata.xml`—Information about the extension and details of the various artifacts in the extension.
- `/Entities/`—Zero or more entity XML files organized in subfolders by entity type.
- `/lib/`—JAR file that includes the custom Java classes for the extension and third-party JAR files required by the extension.
- `/ui/`—Files that define custom widgets used to build and run Mashups.

Bundle Multiple Extensions in One ZIP File for an Application

IoT solutions that contain multiple extensions can be bundled as a single ZIP file. The single ZIP file contains individual ZIP files for each extension. This ensures that every extension is independently implemented. Individual extensions are easier to update in subsequent releases.

Extension Name and Version Convention

There are no restrictions on how you name your extension and the ZIP file of an extension. Note that after you specify a name for the extension, you cannot change it. However, the name of the ZIP file can be changed.

Though you can specify different names for the extension and its ZIP file, it is recommended to have the same name for the extension and its ZIP file. The ZIP file name can also contain the version number.

You can have an IoT solution that contains multiple extensions bundled as a ZIP of ZIPs. In this case, it is recommended to have the same release number for all ZIPs. Consider an IoT solution that contains two extensions. Each extension is a single ZIP file with an individual release number. It is good practice to specify the same release number in the name of the two extensions. Also, specify the same release number in the name of the ZIP of ZIPs. For example, if the release number of your IoT solution is 7.9, specify 7.9 as the release number in the name of all the ZIP files.

In the `metadata.xml` file, the attribute `packageVersion` is used to specify the version of the extension. This is a required attribute, whose value follows the `<major>.<minor>.<patch>` format, where each part of the version is numeric. Extensions must follow the semantic versioning rules. See Semantic Versioning for more information.

Extension Dependencies

This section describes the dependencies of an extension on other extensions.

Dependency of the Extension on Other Extensions

If your extension has a dependency on other extensions, it is recommended to use the `dependsOn` attribute in the `metadata.xml` file to specify the dependencies. The value of the attribute is a comma-separated list of extension names and versions in the `<name>:<major>.<minor>.<patch>` format, where versions are specified only in numbers. For example, `dependsOn= "ExtensionOne:2.5.1,ExtensionTwo:1.2.0"`.

It is a good practice to avoid too many dependencies. If there are upgrades to any of the extensions, dependent extensions are also affected.

Avoid Tight Coupling to Other Extensions

You must avoid tight coupling of your extension to other extensions.

- If your extension is tightly coupled to another extension that must be upgraded to a new major version, you are prompted 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 Things and Thing Templates of the functionality you require. These Things and Thing Templates are not part of your extension, and therefore, are not included in the extension upgrades.
- Occasionally, it can be difficult to avoid tight coupling when you use widgets from other extensions in your extension Mashups.

Size of an Extension

Large and complex extensions can be difficult to maintain and upgrade. A better approach it to split extensions into smaller components based on functionality for easier maintenance and upgrade. You can bundle multiple extensions in a single ZIP file. See the section Bundle Multiple Extensions in One ZIP File for an Application on page 88, for more information.

Use of External JAR Files in an Extension

ThingWorx enables users to include third-party libraries in their code. However, it is recommended that you avoid using the common JAR files. Instead, use the JAR files that are packaged with the SDK in your application. Consider the following:

- The ThingWorx Platform enables multiple sources to use the same JAR files.
- When multiple sources use the same JAR files, a conflict can occur when uploading an extension to the platform, or when an incorrect JAR version is used while building the extension.
- Future versions of the platform may require updates to extensions to continue working.
- A customer cannot use two extensions that require the same JAR files at the same time.

Make Your Entities Non-Editable

When you create entities, such as Mashups, Style Definitions, and so on, make sure that the entities of the extension are non-editable. Only non-editable entities can be upgraded in place. Editable entities cannot be upgraded. By default, new entities are non-editable.

- If you want to have some entities editable, bundle these entities in a separate extension, so that you can easily upgrade the other components of the extension.
- For editable entities, it is recommended to minimize the editable locations on a Mashup. Use embedded Mashups to minimize editable locations.

If you want to provide the ability to customize the extension, for example, adding a custom logo, consider whether an alternative approach can work:

- Can configuration of a Thing be used instead? A non-editable Thing can still have configuration table changes.
- Can a service be used to look up a configuration to provide a media entity or an embedded Mashup?

Upgrade an Extension with Editable Entities

While upgrading an extension with editable entities, perform a migration test to check if any information, such as tags, is lost during the upgrade process.

Organize Entities

It is recommended to group entities using Projects and Model Tags. You must use one Project for all the entities of the extension. Create at least one Model Tag that can be used to tag all the entities of your extension.

Back Up the Storage Before Deleting

It is recommended that you back up your current `ThingworxStorage` folder before deleting it.

Eclipse Plugin

The Eclipse plugin makes it easier to develop and build extensions. It provides actions that automatically generate source files, annotations, and methods. The actions also update the metadata file to ensure proper configuration of the extension. The plugin also ensures that the syntax and format of annotations and the metadata file are correct.

Troubleshooting and Debugging ThingWorx Applications During Development and Packaging

The following section explains how to troubleshoot some issues during the extension development and deployment.

Thing Template Does Not Exist After Successful Import

If there is an issue while creating the Thing Template after importing the entities, the Thing Template may not be available in the application. However, this exception is not fatal. The most common reason for this exception is a missing JAR file that is required for the Thing Template class. For this issue, perform the following checks:

- In the `metadata.xml` file of your extension, check the declaration for the required JAR file.
- Check if the required JAR file is available in the `lib/common` directory of the extension.

JAR File Conflict on Import

You may get the JAR file conflict exception when your application uses a JAR file that is already loaded on the ThingWorx Platform. To fix the issue, you can remove the JAR from your `metadata.xml` file. However, removing the JAR from metadata file can be risky if a different version of JAR is loaded on the platform. Different versions of the same JAR can cause conflicts in the functionality. This conflict could affect your extension or the platform itself. The best solution is to try to avoid using the JAR, if possible.

Check Logs to Debug Issues When Importing an Extension

A ThingWorx application logs all its messages in the application log. When you import an extension, classes are loaded from the JAR files, entities are created, and multiple background processes are executed at the same time. Sometimes this may cause unexpected errors. The import may fail or succeed with some ThingWorx artifacts missing. For such errors, check the application log where you can find the error with an explanation of what went wrong with the import.

To get the application and other logs, use the **Monitoring** menu of the Composer.

Connect a Debug Port to Tomcat

The best way to debug your application is to connect a debug port to your Tomcat instance. This enables you to connect to the platform from an integrated development environment (IDE). You can add breakpoints to the code that you have uploaded. You can trigger a service, set a property, or save a Thing, and track what happens inside your code as it executes. You must upload the same code as the code in which you have breakpoints. The breakpoints you add will be associated with different lines of code and may have different values and functionality.

To add a debug port to Tomcat, you must add a Java option on startup. There are different ways to add a debug port. It depends on how you launch Tomcat. You must configure the port depending on your Tomcat and IDE setup.

Considerations While Upgrading ThingWorx Extensions

Use the following best practices while upgrading the extensions:

Upgrading to ThingWorx 8.4 and Later Releases

JSON libraries are not shipped with ThingWorx 8.4 and later releases. If your extension projects require the JSON libraries, update the `build.gradle` file as below:

- Add the following code in the dependencies block:
  ```
  compile group: 'org.json', name: 'json', version: '20090211'
  ```
- Add the following code anywhere in the gradle file:
  ```
  repositories {
    jcenter()
  }
  ```

Upgrade Java-Backed Extensions

While upgrading Java-backed extensions, use the following best practices:

- When you upgrade a Java-backed extension by importing a new version, restart the platform. When the platform restarts, the extension ZIP file is placed in a queue:

  ```
  /ThingworxStorage/extensions/upgradequeue
  ```

- After you restart the Tomcat server, ThingWorx tries to import the extension ZIP files in the queue.
- Check the application logs after restarting the platform to ensure that all the queued extensions are successfully imported.
- In-place upgrades of complex Java-backed extensions are possible. However, a Tomcat class loader does not allow in-place upgrades of Java-backed extensions in the following scenarios:
  - When you add a new Java-backed entity to your JAR
  - When you delete or rename a Java-backed entity in your JAR

  In such cases, consider the following workaround:

  - You can create new entities if their services are implemented only in JavaScript.
  - It is recommended to create a new extension for the new functionality and further create a separate JAR file.

An Updated Extension Retains the Functionality of an Older Version

The loaded classes from the previous version cause the extension to retain the functionality of the previous version. Restart Tomcat to load the updated extension.

Additional Resources for Packaging and Deploying ThingWorx Applications

This section lists the additional resources that you can reference for a better understanding of the concepts.

Additional Resources for More In-Depth Learning

For more in-depth information on	Link	Resource
Importing extensions	http://support.ptc.com/help/thingworx_hc/thingworx_8_hc/en/index.html#page/ThingWorx/Help/Getting_Started/ImportingandExportingin-ThingWorx/ImportingExtensions.html	Help center
Installing and upgrading ThingWorx	http://support.ptc.com/help/thingworx_hc/thingworx_8_hc/en/index.html#page/ThingWorx/Help/Getting_Started/InstallingandUpgrading/InstallingandUpgrading.html	Help center
Deploying an application	https://developer.thingworx.com/resources/guides/deploy-application-how-to	Developer portal

7

Publishing Applications Built on ThingWorx Platform

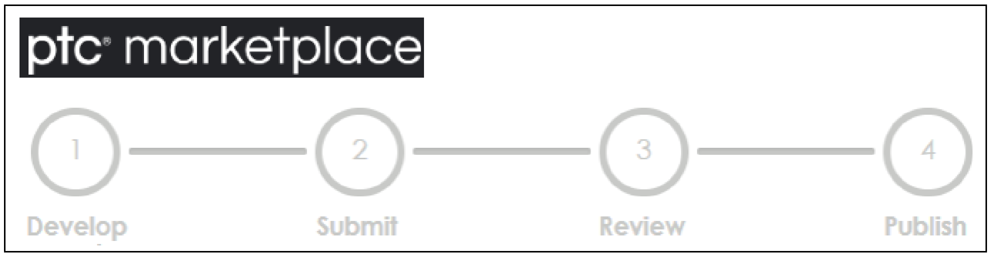
This section provides a summary of the best practices for publishing your applications.

PTC Marketplace

The IoT solutions built on ThingWorx Platform can be published on the PTC Marketplace. Customers can access the Marketplace. On the PTC Marketplace you can publish:

- Tools—Includes extensions, software development tools and starter kits.
  - Extensions are service libraries, connector templates, functional widgets, and more. Extensions enable ThingWorx users to extend the functionality of their IoT applications.
  - Software development tools are used to create applications.
  - Starter kits contain documents that help you connect a device or software product to the ThingWorx Platform. The kits may also include some software components.
- Solutions—Includes commercial and reference solutions.
  - Commercial solutions are complete out-of-the box solutions that are developed by partners.
  - Reference solutions are solutions created for a specific domain that are developed by partners.

The phases of building and publishing your content to the PTC Marketplace are:



1. Develop the IoT application.
2. Submit it to PTC for review.
3. PTC reviews the application.
4. Publish the application to the Marketplace listings.

Best Practices for Publishing ThingWorx Applications

This section provides a summary of the best practices for publishing your applications.

Use the following best practices while publishing your applications to the PTC Marketplace. It is recommended to run virus and security scans on your applications before publishing it to the marketplace.

Activity	Description
Automated Virus Scan	You must check your application for viruses by setting up automated virus scans. Use antivirus applications such as ClamAv to check for potentially harmful viruses.
Static Code Analysis	A static code analysis is a technique used to analyze the software for potentially vulnerable code without executing the code. Use tools such as VeraCode to perform this test.
Dependency Check	You must check your application and its dependent system files such as JARs for vulnerabilities. Use tools such as OWASP ZAP to check the application and its dependencies for components with vulnerabilities.
Security Testing	You must test your application for security loopholes. Use applications, such as Sophos and Sonatype, to perform the security testing.
Functional Testing	You must perform functional testing of your application to ensure that the application conforms with the functional requirements.
Negative Testing	You must perform negative testing to check if your application can handle invalid input data.
Versioning of Extensions	You can publish various versions of your extension on the PTC Marketplace. See the section Extension Name and Version Convention on page 88, for more information.

Additional Resources for Publishing Applications

This section lists the additional resources that you can reference for a better understanding of the concepts.

Additional Resources for More In-Depth Learning

For more in-depth information on	Link	Resource
PTC Marketplace	https://www.ptc.com/en/marketplace/	Marketplace portal
ThingWorx extensibility	http://support.ptc.com/help/thingworx_hc/thingworx_8_hc/en/index.html#page/ThingWorx/Help/Extensibility/ThingWorxExtensibility.html	Help Center

8

Monitoring ThingWorx Applications

Monitoring applications is part of application performance management (APM). This section describes some of the key metrics and performance indicators that help you monitor your ThingWorx applications. It also describes some of the tools that can be deployed to collect and analyze these performance indicators.

It is recommended to monitor the following performance indicators:

- ThingWorx memory usage—Memory issues in Java virtual machine (JVM) or at operating system level are some of the primary reasons for performance issues in applications. They also cause the application to crash.
- ThingWorx Mashup and execution time of services—Slow execution of services or a Mashup impact the user experience of your ThingWorx application.
- Database, operating system, and network performance—ThingWorx depends on database, operating system, and network responsiveness for optimal performance.
- Log monitoring—ThingWorx extensively logs diagnostic messages. These messages should be actively monitored to detect any performance or usability issues.

Monitoring Tools

The ThingWorx platform is developed on Java technology. The diagnostic and monitoring tools explained in this section are designed to collect performance metrics and other data from the Java virtual machine (JVM). Use these tools to monitor your ThingWorx applications through the JVM layer. The tools get the following data from JVM:

- Memory operations—Check if the server has enough memory resources allocated, and how long the JVM Garbage Collection (GC) operations take.
- Thread performance—Check if there are any long transactions or threading issues that could cause performance issues in your ThingWorx application.

PTC System Monitor (PSM)

PTC System Monitor (PSM) is an application performance monitoring system powered by Dynatrace. PTC has customized Dynatrace for the monitoring requirements of ThingWorx applications. PSM provides system administrators with tools to detect, diagnose, and repair problems that impact the operational health of the application. It monitors both the test and production systems. It does not impact the performance of the ThingWorx application. This module is available as part of your maintenance contract and does not require the purchase of additional licenses for the core platform monitoring.

PSM monitors the thread-level activity that occurs at the JVM level and distills this activity into a proprietary technology called PurePaths. PSM PurePath contains details about the execution path of the ThingWorx service on the server and the execution time. It also contains context information, such as executed SQL statements.

Capabilities of PSM

PSM enables you to:

- Increase the up-time of the application before it fails:
  - Sends notifications when the thresholds are breached, for example, high usage of CPU. This enables the administrator to take corrective actions to address the issues before it turns into a system-wide problem.
  - Provides a historic view of the usage of system resources. This helps you estimate the future capacity requirements for system resources.
- Improve the monitoring capabilities:
  - Uses simplified and customizable dashboards to summarize the health of the system and components.
  - Provides drill-down capabilities to individual transactions, users, and server components for more details.
  - Contains an integrated incident list for issues that impact health of the system.

- Improve the diagnostic capabilities with data in real-time:
  - Uses the PurePath technology to gather diagnostic data in real-time and record historical performance metrics.
  - Eliminates hours spent in reproducing issues with additional verbose settings and troubleshooting issues with PTC Technical Support.

PSM continuously collects and records real-time data from your ThingWorx application and stores this data for extended periods of time. The data collected by PSM enables you to continuously monitor the following performance metrics that impact custom applications. This helps your development team or PTC Support Services to replay the events that lead to the performance issue:

- JVM memory performance
- Service execution times using PurePath time data
- Operating system memory, CPU, and database query metrics
- ThingWorx log errors and other incidents

Direct JVM Monitoring

Direct JVM monitoring tools are either provided by Oracle or ThingWorx Support Tools extension. These tools can be used along with PSM to obtain detailed performance data. If you do not use PSM, collecting data directly from the JVM is a good alternative to identify and fix performance issues. It is recommended that you monitor the following JVM diagnostics for your ThingWorx applications:

- Monitor the memory performance over time using Garbage Collection logging, an in-built capability of JVM.
- Thread-level collection and analysis using the ThingWorx Support Tools extension.

Monitor the Memory Usage of a ThingWorx Server

The Java Virtual Machine (JVM) manages its own memory (heap) natively using the Garbage Collector (GC). GC identifies and removes objects in the Java heap that are not in use. You can monitor the memory usage of a ThingWorx server by logging the details collected by the GC.

You can use the GC log file to identify trends in the memory consumption. Depending on the trends, you can check if the maximum heap parameter of the Apache Tomcat server should be changed for better performance. Therefore, it is recommended to set up logging for GC on the Apache Tomcat server.

JVM has flags that are called at runtime. These flags are used to write the statistics about GC events, such as the type of GC event, amount of memory consumed at the start of the event, amount of memory released by the GC event, and duration of the GC event.

The GC log file is overwritten every time the JVM is started. It is recommended that you back up the log file during the server restart. This helps you analyze if the memory usage caused the server to restart.

You can use analysis tools such as, GCEasy.io and Chewiebug GC Viewer, to analyze the logs from Garbage Collection.

How to Set Up Garbage Collection Logging on Linux (setenv.sh)

Perform the following steps to set up Garbage Collector logging on Linux:

1. Open the `$CATALINA_HOME/bin/setenv.sh` script in a text editor.

   The `setenv.sh` script file may not be available in all installations. If the file does not exist, create a new `setenv.sh` file at the location.
2. Append the following text at the end of the `JAVA_OPTS` variable within the quotes.
   ```
   -Xloggc:$CATALINA_HOME/logs/gc.out -XX:+PrintGCTimeStamps
          -XX:+PrintGCDateStamps -XX:+PrintGCDetails
   ```
   Note the following:
   - The `-XX:+PrintGC` option returns a less verbose output log. For example, the output log has the following details:
     ```
     2017-10-10T13:22:49.363-0400: 3.096: [GC (Allocation
     Failure)
                         116859K->56193K(515776K), 0.0728488
     secs]
     ```
   - The `-XX:+PrintGCDetails` option returns detailed output log. For example, the output log has the following details:
     ```
     2017-10-10T13:18:36.663-0400: 35.578: [GC (Allocation Failure)
       2017-10-10T13:18:36.663-0400: 35.578: [ParNew: 76148K->6560K(76672K),
     0.0105080 secs]
       262740K->193791K(515776K), 0.0105759 secs] [Times: user=0.02 sys=
     0.00, real=0.01 secs]
     ```
   - If `JAVA_HOME` is not specified in the `setenv.sh` file or the `setenv.sh` file is new, add the following line in the file:
     ```
     JAVA_HOME=”-Xloggc:$CATALINA_HOME/logs/gc.out -XX:
     +PrintGCTimeStamps
           -XX:+PrintGCDateStamps -XX:+PrintGCDetails”
     ```
3. Add the following lines at the end of the `setenv.sh` script. This code automatically backs up an existing `gc.out` file to `gc.out.restart` before starting the Apache Tomcat server:
   ```
   # Backup the gc.out file to gc.out.restart when a server is
   started.
   if [ -e "$CATALINA_HOME/logs/gc.out" ]; then
       cp -f "$CATALINA_HOME/logs/gc.out" "$CATALINA_HOME/logs/
   gc.out.restart"
   ```

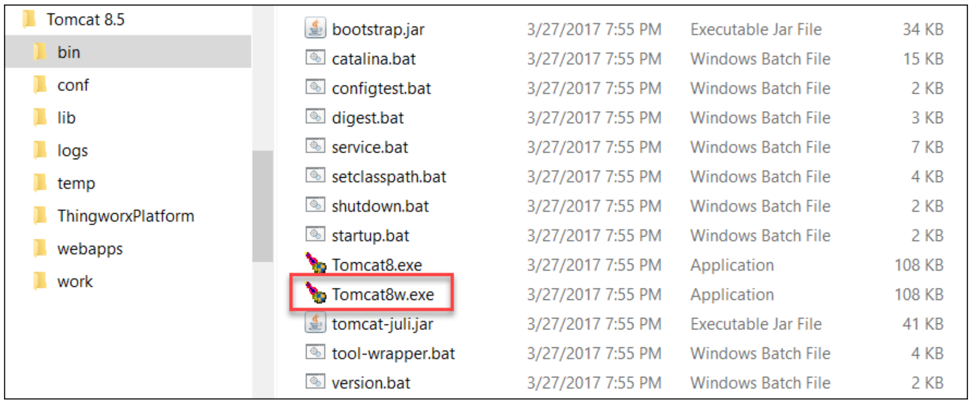
```
fi
```

4. Save the changes to the file.
5. Restart the Apache Tomcat service using the service controller that is applicable to your operating system.

How to Set Up Garbage Collection Logging on Windows (Apache Service Manager)

Perform the following steps to set up Garbage Collector logging on Windows:

1. In Windows Explorer, browse to `<Tomcat_home>\bin`.
2. Start the executable that has `w` in its name. For example, `Tomcat<version number>w.exe`.



Tomcat 8.5
- bin
- conf
- lib
- logs
- temp
- ThingworxPlatform
- webapps
- work

bootstrap.jar	3/27/2017 7:55 PM	Executable Jar File	34 KB
catalina.bat	3/27/2017 7:55 PM	Windows Batch File	15 KB
configtest.bat	3/27/2017 7:55 PM	Windows Batch File	2 KB
digest.bat	3/27/2017 7:55 PM	Windows Batch File	3 KB
service.bat	3/27/2017 7:55 PM	Windows Batch File	7 KB
setclasspath.bat	3/27/2017 7:55 PM	Windows Batch File	4 KB
shutdown.bat	3/27/2017 7:55 PM	Windows Batch File	2 KB
startup.bat	3/27/2017 7:55 PM	Windows Batch File	2 KB
Tomcat8.exe	3/27/2017 7:55 PM	Application	108 KB
Tomcat8w.exe	3/27/2017 7:55 PM	Application	108 KB
tomcat-juli.jar	3/27/2017 7:55 PM	Executable Jar File	41 KB
tool-wrapper.bat	3/27/2017 7:55 PM	Windows Batch File	4 KB
version.bat	3/27/2017 7:55 PM	Windows Batch File	2 KB

3. Click the **Java** tab.
4. In the **Java Options** field, add the following lines:
   ```
   -Xloggc:logs/gc_%t.out
   -XX:+PrintGCTimeStamps
   -XX:+PrintGCDateStamps
   -XX:+PrintGCDetails
   ```
   For information on the between `-XX:+PrintGC` and `-XX:+PrintGCDetails` options, see the section *How to Set Up Garbage Collection Logging on Linux (setenv.sh)*.
5. Click **Apply** to save the changes to the service configuration.
6. In the **General** tab, click **Stop**.
7. Click **Start** to restart the Apache Tomcat service.

   The new garbage collection log is written to `%CATALINA_HOME%\logs\gc_<Tomcat_Restart_Timestamp>.out` file.

Collect Diagnostic Data Using Support Tools Extension

The ThingWorx Support Tools extension is used to collect information for ThingWorx Technical Support. This information is used while troubleshooting problems in your ThingWorx application. The extension collects the following diagnostic data:

- Java thread dumps—Details about all the threads that are running on a Java virtual machine at a specific time. It can record the thread activity for specified periods.
- Java heap dumps—Details about all the objects in the heap space of a Java virtual machine at a specific time.

PTC recommends that you install this extension on your ThingWorx instance. The extension and the User Guide documentation are available at the PTC Marketplace.

VisualVM and Other JMX Monitoring Tools

VisualVM is a monitoring tool that collects data for Java applications. You can use VisualVM to monitor ThingWorx applications. It collects information about memory and CPU usage. It generates and analyzes heap dumps and tracks memory leaks. You can collect data for applications that are run locally or on remote machines.

VisualVM provides an interface that enables you to graphically view the information about Java applications.

VisualVM is available along with Java JDK. See the section *Apache Tomcat Java Option Settings* in ThingWorx Help Center for more information on connection settings for VisualVM.

Other tools that integrate with the JVM using Java Management Extensions (JMX) also provide similar monitoring capabilities.

VisualVM captures the following key metrics that impact performance of the application:

- JVM memory performance
- Time required to execute threads
- Memory of operating system and CPU metrics
- Database connection pool monitoring

VisualVM tracks real-time data for approximately 20 minutes.

ThingWorx Application Logs Monitoring

ThingWorx application logs should be monitored regularly for errors or other abnormal operations. Errors can occur either at the platform layer, or in the custom scripts used with ThingWorx. It is recommended to review messages in the error log daily.

Configure the LoggingSubsystem Option for Writing Stack Trace for Errors

The **Enable Stack Tracing** option in **LoggingSubsystem** writes error messages and associated stack traces to log files. By default, **LoggingSubsytem** is not configured to write stack traces (call stacks) to disk. Set the **Enable Stack Tracing** option to write call stacks of exceptions and error messages to the `ErrorLog.log` file that is available in the `ThingworxStorage/logs` folder. It is recommended to set this option to have the details of function calls in the stack trace while debugging an error.

Retain the Application Logs

The log level determines how much granularity should be displayed in the logs. It is recommended to keep the log level of the application to minimum verbosity such as **Info** or **Warn**, unless you are actively troubleshooting an issue. Be careful when you increase the log verbosity to **Debug**, **Trace**, or **All**. This can have an adverse effect on the performance of ThingWorx. It can cause unpredictable behavior of the application if the resources available on the platform are insufficient.

Logs are saved as separate files on the server in the `/ThingworxStorage/logs` folder. The logs are archived in the `archives` folder located in the `logs` folder. The log rotation rules are based on the file size and time configurations that are set in the **Log Retention Settings** of the logging subsystem. These settings can be changed and applied at runtime.

You can specify the following rollover configurations in **System ▸ Subsystems ▸ LoggingSubsystem ▸ Configuration**:

- File size—The log file is archived when its size reaches or exceeds the size defined in the **Maximum File Size In KB** field. The default file size for a log is set to 100000 KB. The maximum file size that you can set is 1000000 KB.
- Time—The files are rolled over daily at midnight, when a log event is triggered. The logs are moved to the `archives` folder. By default, if a log is in the `archives` folder for more than seven days, it is deleted. The default can be changed in the **Maximum Number Days For Archive** field. You can specify 1 to 360 days.

It is important to monitor the log sizes. Purge the log files consistently by backing up the files first or discarding them when they contain obsolete information.

Monitoring and Troubleshooting Performance Issues

This section describes how to monitor the following performance areas using the monitoring tools. It also explains how to troubleshoot common performance issues. Use the following best practices to ensure that your ThingWorx application remains responsive during performance issues:

- Memory performance on page 106
- Slow service executions on page 115
- Operating system and database performance on page 127
- Network and connectivity performance on page 132

Memory Performance

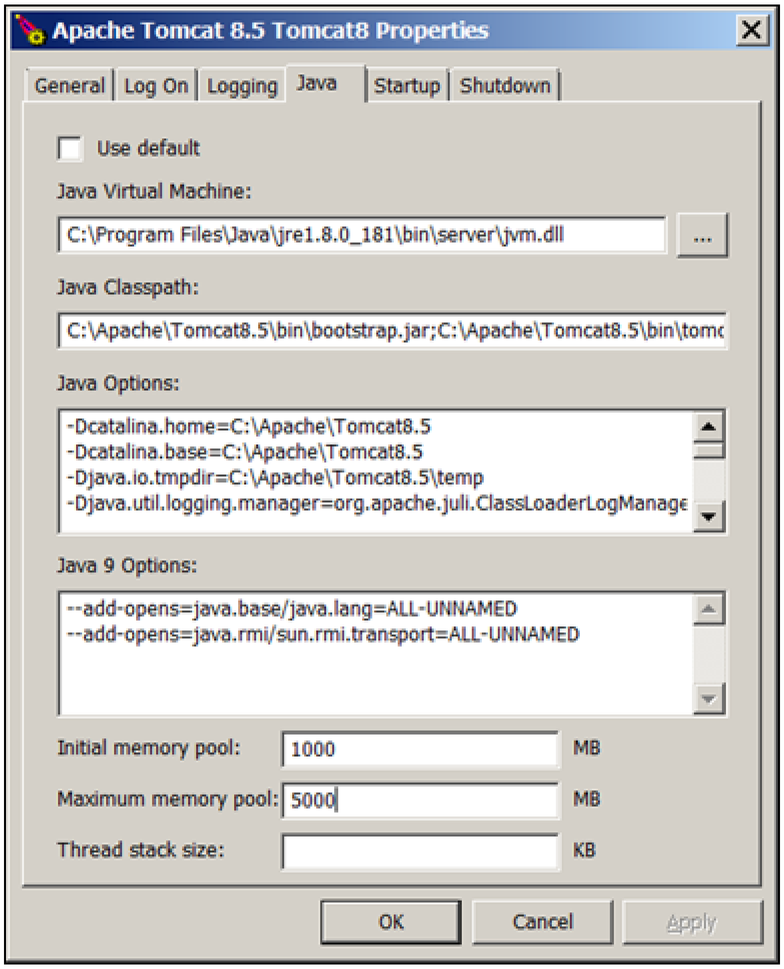
Performance issues are generally seen in the overall memory utilization of ThingWorx applications. If your application uses the Java memory heap aggressively and constantly creates and discards large memory objects, you see some poor performance indicators in the memory metrics for this system. The frequency and duration of Java garbage collections play a significant role in the overall responsiveness of the system. If you have continuous garbage collection loops, the application is stressed for memory, and this can make your entire ThingWorx platform entirely unresponsive.

To monitor Java memory usage and garbage collection over time, it is recommended to enable GC logging. Tools such as PSM or VisualVM also display memory utilization information. However, if the JVM is unresponsive due to full GC loops, these external applications may not be able to retrieve data to show the memory issues.

Note that when you monitor memory at operating system level, it does not provide information on how your ThingWorx application uses the memory allocated to it. The monitoring tool retrieves data on how much memory Java requested from the operating system, but it does not show how Java uses the memory. Most performance issues are due to the internal memory usage in the Java heap space, not the operating system memory allocations.

The overall memory allocated to Java is determined by the `-Xms` (minimum) and `-Xmx` (maximum) parameters that are loaded at runtime. The `-Xms` and `-Xmx` parameters specify the initial and maximum memory pool size, if Tomcat is configured using the `tomcat8w` configuration panel.

If no minimum or maximum values are specified at the start up, the Java process at startup uses a heuristic to preallocate these amounts. By default, Java attempts to allocate one-fourth of the overall RAM available in the operating system.



Apache Tomcat 8.5 Tomcat8 Properties

Use default: ☐

Java Virtual Machine: C:\Program Files\Java\jre1.8.0_181\bin\server\jvm.dll

Java Classpath: C:\Apache\Tomcat8.5\bin\bootstrap.jar;C:\Apache\Tomcat8.5\bin\tomc

Java Options:

```
-Dcatalina.home=C:\Apache\Tomcat8.5
-Dcatalina.base=C:\Apache\Tomcat8.5
-Djava.io.tmpdir=C:\Apache\Tomcat8.5\temp
-Djava.util.logging.manager=org.apache.juli.ClassLoaderLogManage
```

Java 9 Options:

```
--add-opens=java.base/java.lang=ALL-UNNAMED
--add-opens=java.rmi/sun.rmi.transport=ALL-UNNAMED
```

Initial memory pool: 1000 MB

Maximum memory pool: 5000 MB

Thread stack size: KB

Operations That Cause Memory Issues

You must monitor the following operations that can cause memory issues:

- Extremely long and frequent full GC operations with the duration of 45+ seconds—Full GCs are undesirable, as they are stop-the-world operations. The longer the JVM pauses, the more are the memory issues:
  - All other active threads are stopped while the JVM attempts to make more memory available.
  - Full GCs take considerable time, sometimes minutes, during which the application may become unresponsive.
- Full GCs occurring in a loop—Full GCs occurring in quick succession are called full GC loops. They can cause the following issues:
  - If the JVM is unable to clean up any additional memory, the loop can go on indefinitely.
  - The ThingWorx application is unavailable to users while the loop continues.

- Memory leaks—Memory leaks occur in the application when an increasing number of objects are retained in the memory. These objects cannot be cleaned, regardless of the type of garbage collection the JVM performs.
  - When you check for memory consumption over time, a memory leak appears as an ever-increasing memory usage that was never reclaimed.
  - The server eventually runs out of memory, regardless of the upper bounds set.

How to Monitor Memory Issues

On a production system, it is recommended to monitor overall memory usage on a daily basis either manually using the GC log file, or automatically using tools, such as PSM or VisualVM.

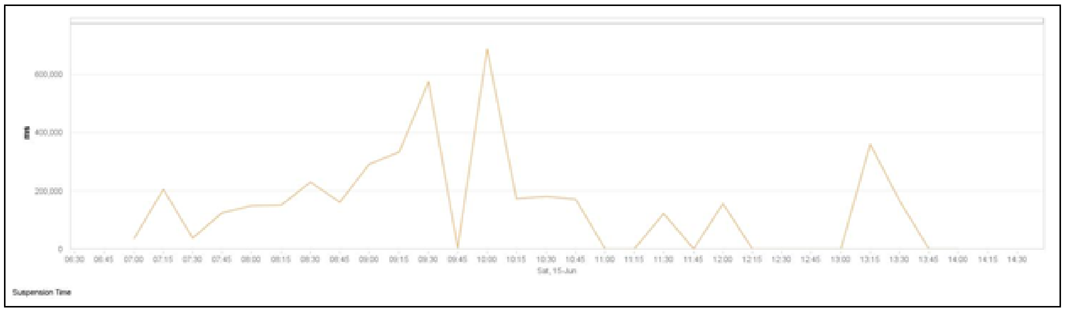
You can monitor the memory usage using any of the following tools. You can use one or more tools together to get a holistic view on the memory usage in an application.

- PSM metrics and graphs
- GC log analysis
- Java VisualVM
- Heap dumps
- Tomcat Apache logs

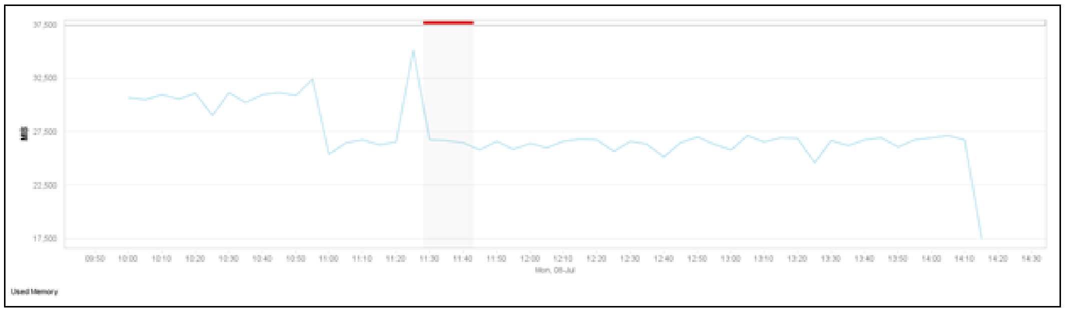
Using PSM Metrics to Monitor Memory Issues

PMS tracks the time duration for which JVM was unresponsive due to garbage collection. The amount of time spent in suspension (not actively performing work) is determined by creating a custom graph, which is available at the **Charting ▸ Custom ▸ Server Side Performance ▸ Agent Based Measures ▸ Java Virtual Machine ▸ Suspension Time** option.

With the recommended Garbage First (G1) collector, the JVM suspends activity for a maximum of 200 ms at a time. Generally, ThingWorx remains responsive with pauses in the range of 200 ms to 5000 ms. The following graph shows scenarios where the entire application is unresponsive for long periods.



Usually, periods of slow GC activity correspond to periods of high memory usage. The memory usage, shown at the operating system level does not retrieve data on how much memory Java is using internally to its heap. You can get this information in PSM using the **Charting** ▸ **Custom** ▸ **Server Side Performance** ▸ **Agent Based Measures** ▸ **Java Virtual Machine** ▸ **Used Memory** option.



When Java consistently uses high memory internally, it can become a performance bottleneck. PSM is used to drill-down into any period of high GC suspension or high memory usage to identify the underlying user transactions at that time. Long PurePaths that correspond to memory issues should be investigated.

PSM creates an incident if the application is in out-of-memory state. For any out-of-memory condition, PSM generates the mentioned two graphs to help identify when the memory usage spike occurred. You can drill-down into specific transactions by analyzing the PurePaths that were executed at that time.

Recommendation—Set up the PSM alerting or a daily schedule to check JVM suspension time and used memory. Monitor for any out-of-memory incidents in PSM.

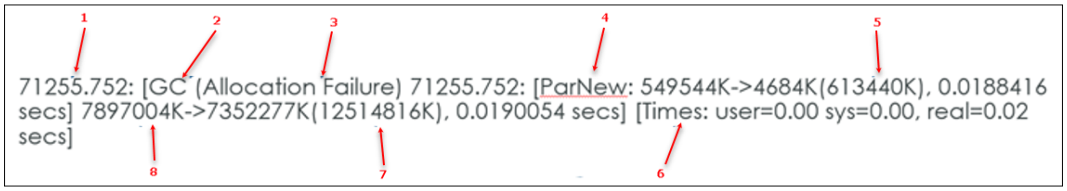
Analyzing the Log Files of Garbage Collector (GC) to Monitor Memory Issues

It is recommended that you enable GC logging in production environments. See the section Monitor the Memory Usage of a ThingWorx Server on page 101, for more information on how to enable GC logging. After GC logging is enabled, all the GC activity on the server is logged. If the JVM is unresponsive for long periods of time, external tools, such as PSM or VisualVM, may not be able to collect memory data. However, JVM can print the GC activity, which helps you identify the cause for the memory issue.

Reading the GC Logs

You can use tools such as, GCEasy.io and Chewiebug GC Viewer, to analyze GC logs. You can also analyze GC logs manually in a text editor for common issues.

Each memory cleanup operation is generally printed, as shown in the following example.

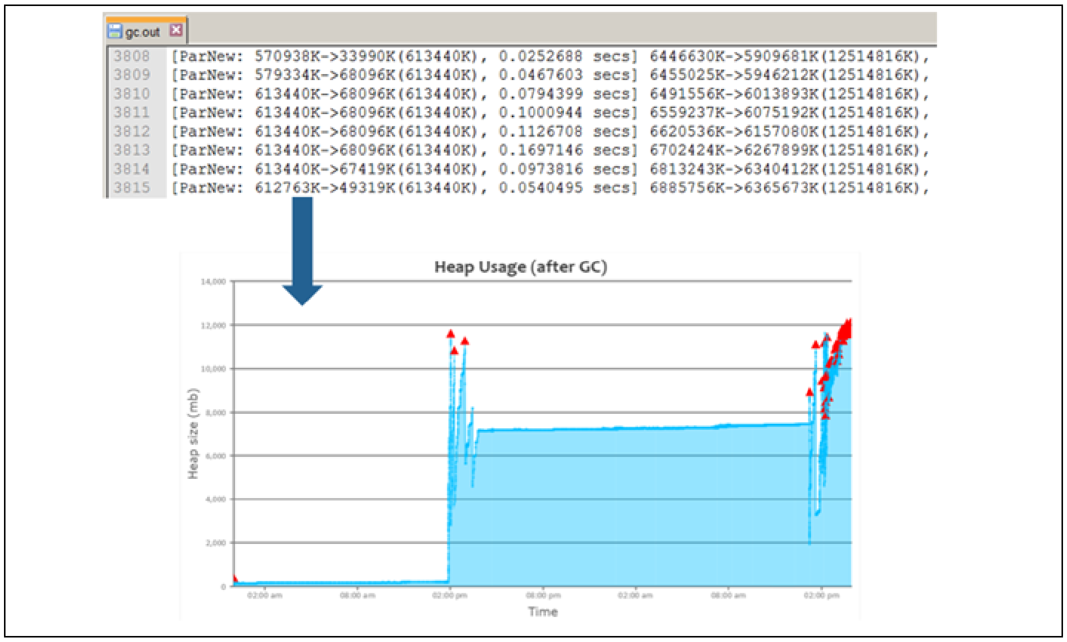


```
71255.752: [GC (Allocation Failure) 71255.752: [ParNew: 549544K->4684K(613440K), 0.0188416
secs] 7897004K->7352277K(12514816K), 0.0190054 secs] [Times: user=0.00 sys=0.00, real=0.02
secs]
```

1 Time since JVM start
2 Type of GC
3 Reason for GC
4 Young Generation cleaned up from 549M to 4M
5 Total Young Generation size
6 Overall time for GC to complete
7 Heap allocated
8 Total memory of the application that Java is cleaning

See Oracle documentation for more information on GC operations.

A GC analysis tool converts the text representation of the cleanups into a more easily readable graph.



```
3808  [ParNew: 570938K->33990K(613440K), 0.0252688 secs] 6446630K->5909681K(12514816K),
3809  [ParNew: 579334K->68096K(613440K), 0.0467603 secs] 6455025K->5946212K(12514816K),
3810  [ParNew: 613440K->68096K(613440K), 0.0794399 secs] 6491556K->6013893K(12514816K),
3811  [ParNew: 613440K->68096K(613440K), 0.1000944 secs] 6559237K->6075192K(12514816K),
3812  [ParNew: 613440K->68096K(613440K), 0.1126708 secs] 6620536K->6157080K(12514816K),
3813  [ParNew: 613440K->68096K(613440K), 0.1697146 secs] 6702424K->6267899K(12514816K),
3814  [ParNew: 613440K->67419K(613440K), 0.0973816 secs] 6813243K->6340412K(12514816K),
3815  [ParNew: 612763K->49319K(613440K), 0.0540495 secs] 6885756K->6365673K(12514816K),
```

Analyzing the data in the GC log helps you identify the following scenarios that indicate memory issues in your ThingWorx application:

- Extremely long full GC operations (45 seconds+)—These are stop-the-world operations, which are highly undesirable. For example, the following full GC takes 46 seconds. During this time all other user activity is stopped:

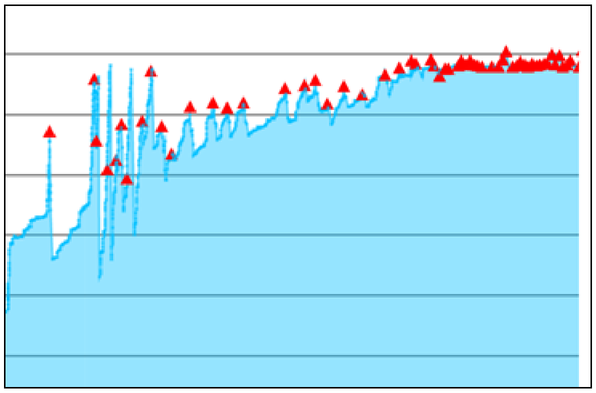
  ```
  272646.952: [Full GC: 7683158K->4864182K(8482304K) 46.204661 secs]
  ```

- Full GCs occurring in a loop—These are full GCs occurring in a quick succession.

For example, the following four consecutive garbage collections take nearly two minutes to resolve:

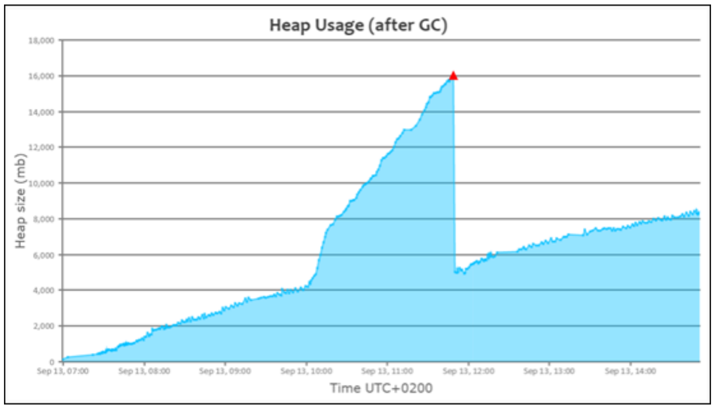
```
16121.092: [Full GC: 7341688K->4367406K(8482304K), 38.774491
secs]
16160.11:  [Full GC: 4399944K->4350426K(8482304K), 24.273553
secs]
16184.461: [Full GC: 4350427K->4350426K(8482304K), 23.465647
secs]
16207.996: [Full GC: 4350431K->4350427K(8482304K), 21.933158
secs]
```

- In the following graph, the red triangles on the graph spikes are visual indications that full GCs are occurring in a loop:



- Memory leaks—These occur when an increasing number of objects are retained in memory, and the memory cannot be cleaned.

  In the following example, the memory is increasing steadily despite the garbage collectors:

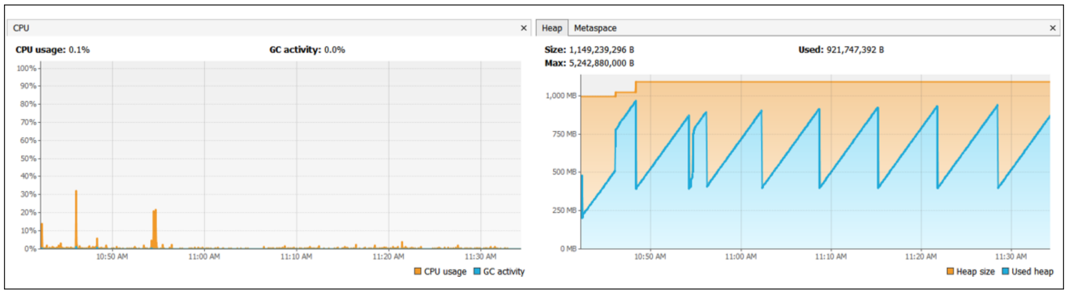


Recommendation—If there is no automated tool, such as PSM, to monitor and record memory usage over time, then check the GC log files daily for memory issues. When memory issues occur, collect JVM thread dumps or heap dumps to identify long running operations. These operations can trigger the underlying memory issues in your ThingWorx application.

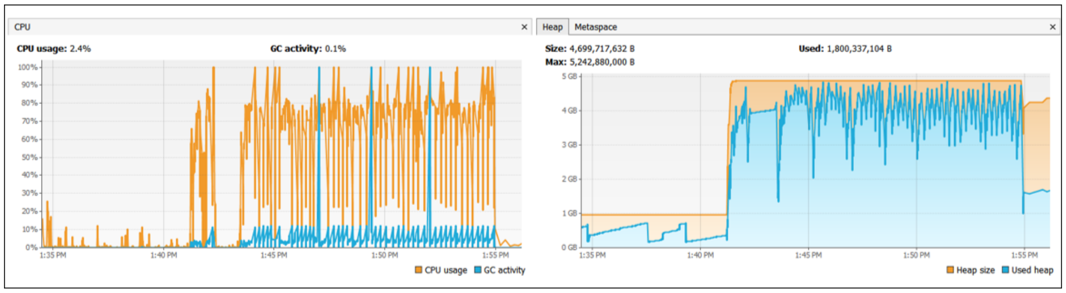
Using VisualVM to Monitor Memory Issues

When VisualVM is connected to your ThingWorx application, you can retrieve real-time data that shows the memory, CPU, and thread usage in your application. VisualVM is used to identify common memory issues and to further drill-down into the root cause of the behavior.

When your ThingWorx application is in a good state, VisualVM shows steady memory and CPU usage over time. The used memory graph shows regular series of peaks. This is because JVM uses the available memory before performing a cleanup. The GC activity is a negligible percentage (less than 5%) of the total CPU usage.

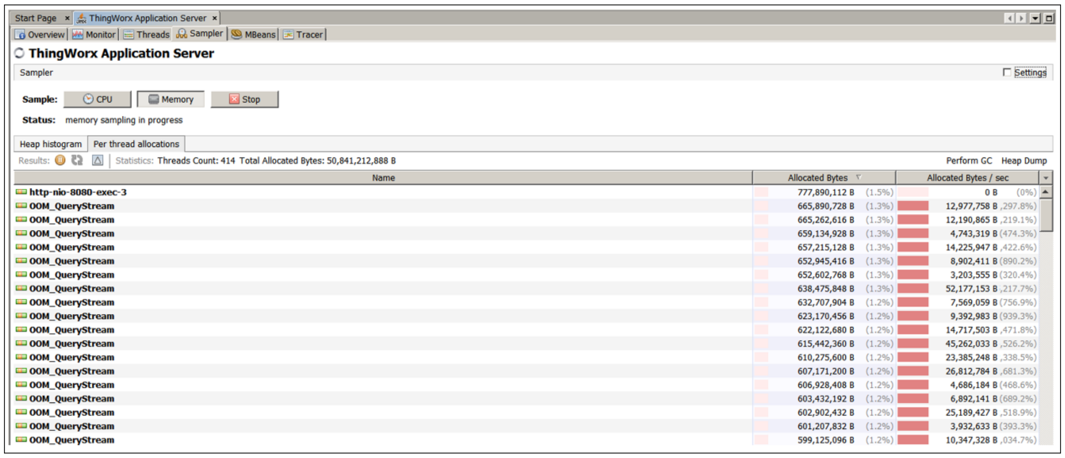


When the system requires more resources, the used memory levels remain elevated for long periods of time. There is a constant or sometimes extremely high GC activity as a percentage of total CPU usage.



The graphical indications of memory exhaustion are similar to the memory metrics of other tools, such as PSM or the GC log. When you use VisualVM, you do not get a specific metric at the JVM level. It only gives a visual indication of the problem. To identify the root cause of the issue, you should analyze the problem in detail.

When you see memory issues in VisualVM, measure which threads are using the memory the most. Check this in the **Sample ▸ Memory ▸ Per thread allocations** option.



ThingWorx Application Server

Sampler

Sample: CPU | Memory | Stop

Status: memory sampling in progress

Heap histogram | Per thread allocations

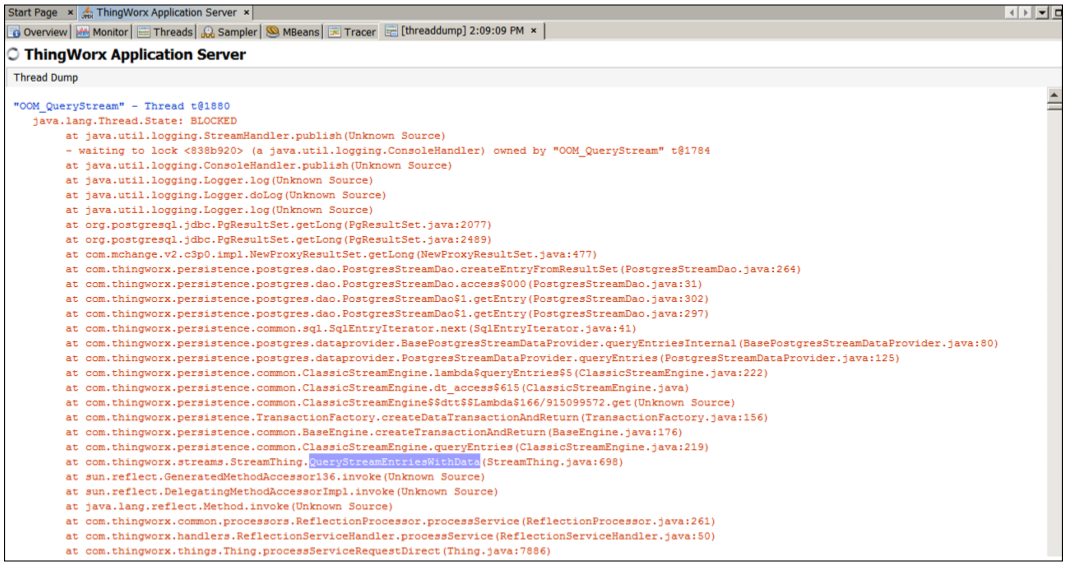
Statistics: Threads Count: 414 Total Allocated Bytes: 50,841,212,888 B

Name	Allocated Bytes	Allocated Bytes / sec
http-nio-8080-exec-3	777,890,112 B (1.5%)	0 B (0%)
OOM_QueryStream	665,890,728 B (1.3%)	12,977,758 B ,297.8%)
OOM_QueryStream	665,262,616 B (1.3%)	12,190,865 B ,219.1%)
OOM_QueryStream	659,134,928 B (1.3%)	4,743,319 B (474.3%)
OOM_QueryStream	657,215,128 B (1.3%)	14,225,947 B ,422.6%)
OOM_QueryStream	652,945,416 B (1.3%)	8,902,411 B (890.2%)
OOM_QueryStream	652,602,768 B (1.3%)	3,203,555 B (320.4%)
OOM_QueryStream	638,475,848 B (1.3%)	52,177,153 B ,217.7%)
OOM_QueryStream	632,707,904 B (1.2%)	7,569,059 B (756.9%)
OOM_QueryStream	623,170,456 B (1.2%)	9,392,983 B (939.3%)
OOM_QueryStream	622,122,680 B (1.2%)	14,717,503 B ,471.8%)
OOM_QueryStream	615,442,360 B (1.2%)	45,262,033 B ,526.2%)
OOM_QueryStream	610,275,600 B (1.2%)	23,385,248 B ,338.5%)
OOM_QueryStream	607,171,200 B (1.2%)	26,812,784 B ,681.3%)
OOM_QueryStream	606,928,408 B (1.2%)	4,686,184 B (468.6%)
OOM_QueryStream	603,432,192 B (1.2%)	6,892,141 B (689.2%)
OOM_QueryStream	602,902,432 B (1.2%)	25,189,427 B ,518.9%)
OOM_QueryStream	601,207,832 B (1.2%)	3,932,633 B (393.3%)
OOM_QueryStream	599,125,096 B (1.2%)	10,347,328 B ,034.7%)

Note

The sampling data in a production environment has additional performance impact and can cause a slow application to become entirely unresponsive. In production environments, use the sampler as the last alternative to obtain diagnostic data.

When you enable the sampler option in VisualVM, you can check which threads use the most memory in your ThingWorx application. To see what these threads are doing, stop the sampler, collect the thread dump from the **Threads** tab, and search the result for the top thread names. For example, in the preceding image, the `OOM_QueryStream` threads perform the `QueryStreamEntriesWithData` lookup, which are known to trigger memory issues in ThingWorx applications depending on how this service is called.



```
"OOM_QueryStream" - Thread t@1880
   java.lang.Thread.State: BLOCKED
        at java.util.logging.StreamHandler.publish(Unknown Source)
        - waiting to lock <838b920> (a java.util.logging.ConsoleHandler) owned by "OOM_QueryStream" t@1784
        at java.util.logging.ConsoleHandler.publish(Unknown Source)
        at java.util.logging.Logger.log(Unknown Source)
        at java.util.logging.Logger.doLog(Unknown Source)
        at java.util.logging.Logger.log(Unknown Source)
        at org.postgresql.jdbc.PgResultSet.getLong(PgResultSet.java:2077)
        at org.postgresql.jdbc.PgResultSet.getLong(PgResultSet.java:2489)
        at com.mchange.v2.c3p0.impl.NewProxyResultSet.getLong(NewProxyResultSet.java:477)
        at com.thingworx.persistence.postgres.dao.PostgresStreamDao.createEntryFromResultSet(PostgresStreamDao.java:264)
        at com.thingworx.persistence.postgres.dao.PostgresStreamDao.access$000(PostgresStreamDao.java:31)
        at com.thingworx.persistence.postgres.dao.PostgresStreamDao$1.getEntry(PostgresStreamDao.java:302)
        at com.thingworx.persistence.postgres.dao.PostgresStreamDao$1.getEntry(PostgresStreamDao.java:297)
        at com.thingworx.persistence.common.sql.SqlEntryIterator.next(SqlEntryIterator.java:41)
        at com.thingworx.persistence.postgres.dataprovider.BasePostgresStreamDataProvider.queryEntriesInternal(BasePostgresStreamDataProvider.java:80)
        at com.thingworx.persistence.postgres.dataprovider.PostgresStreamDataProvider.queryEntries(PostgresStreamDataProvider.java:125)
        at com.thingworx.persistence.common.ClassicStreamEngine.lambda$queryEntries$5(ClassicStreamEngine.java:222)
        at com.thingworx.persistence.common.ClassicStreamEngine.dt_access$615(ClassicStreamEngine.java)
        at com.thingworx.persistence.common.ClassicStreamEngine$$dtt$$Lambda$166/915099572.get(Unknown Source)
        at com.thingworx.persistence.TransactionFactory.createDataTransactionAndReturn(TransactionFactory.java:156)
        at com.thingworx.persistence.common.BaseEngine.createTransactionAndReturn(BaseEngine.java:176)
        at com.thingworx.persistence.common.ClassicStreamEngine.queryEntries(ClassicStreamEngine.java:219)
        at com.thingworx.streams.StreamThing.QueryStreamEntriesWithData(StreamThing.java:698)
        at sun.reflect.GeneratedMethodAccessor136.invoke(Unknown Source)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
        at java.lang.reflect.Method.invoke(Unknown Source)
        at com.thingworx.common.processors.ReflectionProcessor.processService(ReflectionProcessor.java:261)
        at com.thingworx.handlers.ReflectionServiceHandler.processService(ReflectionServiceHandler.java:50)
        at com.thingworx.things.Thing.processServiceRequestDirect(Thing.java:7886)
```

When there is problem, you can collect a snapshot of the application. The snapshot captures all the graphs and thread dumps. This data can be provided to PTC Support for additional analysis of the issue. Collecting a snapshot does not impact the performance of the application.

VisualVM also allows you to perform GC operations manually and to collect heap dumps. These operations should not be performed unless advised by PTC Support.

Recommendation—In case of periods of slow performance, it is recommended to inspect the memory graphs in VisualVM and sample the memory to identify periods of high usage.

Using Other Tools to Monitor Memory Issues

Catalina logs and application logs should be monitored regularly for any out-of-memory error messages. If possible, additional tools such as PSM or VisualVM should be deployed to collect real-time diagnostic data from the system at the exact time corresponding to these errors.

Heap dumps can also be collected using VisualVM, the Support Tools extension, or the `jmap` command that is available with Java Development Kit (JDK). If you collect heap dump for a server that has significant memory allocated to it, the server may become non-responsive for up to 30 minutes during the collection. You must collect the heap dump when advised by PTC Support. Heap dumps can be sent to PTC for analysis, or PTC support can help in analyzing the data collected locally. If a heap dump is collected at the right time, it is the most definitive method of identifying contributors to high memory consumptions. However, this requires analyzing the collected data, which is not in the scope of this guide.

Best Practices for Reducing Memory Issues

When you develop and monitor your ThingWorx applications, use the following best practices to avoid memory-driven performance issues:

- Monitor the daily usage of memory in your ThingWorx application functions. This helps you understand the memory usage in your application in steady state and under stress-conditions.
- Perform scalability testing with production-like data using Apache JMeter or similar utilities to model many users and devices interacting with your ThingWorx application at the same time.

  You must ensure that enough system resources are allocated for scalability testing. Most production systems require 64 GB of RAM, out of which 40 GB is allocated to Java heap.
- Stream queries can retrieve large number of records from the database before they are further filtered in the JVM memory. It is recommended to add appropriate time intervals to minimize the number of rows retrieved by streams. This helps especially when data starts to accumulate on the daily production use.
- If possible, use external systems, such as a database server, microservice server, or a federated ThingWorx server for heavy in-memory processing.
- Optimize long running services. Any service running for over 10 minutes is likely to lock memory, database connections, and other server resources during execution. The scheduled services should also be executed and completed in under 10 minutes to ensure memory resources are returned to the system.
- Consider implementing a high availability cluster for production usage. This ensures that the overall application remains available even if one of the ThingWorx nodes encounters memory issues.

Slow Execution of Services

Slow execution of services impacts the users of your ThingWorx application. They have to wait for considerably long time for the service to complete before they see the results in the user interface. If many services are executed slowly and simultaneously, your ThingWorx application can become unresponsive.

This section explains some of the common indicators of performance issues in the services layer of your ThingWorx application. The section helps you understand which metrics to monitor, how the internal subsystems handle various operations, and how bottlenecks in these subsystems impact the overall server performance.

Operations That Cause Slow Execution of Services

You must consider the following while monitoring slow execution of services:

- How quickly are the services executed? Are there long-running services that should be optimized?
- Are there any blocked services?
- Are there services that can cause contention on shared resources such as database connections?
- Are there any ThingWorx subsystems that are delayed on processing? For example, are the schedulers or timers taking too long to complete the operation and causing a bottleneck in the Event Subsystem?

How to Monitor Slow Execution of Services

You can use the following tools to identify slow operations, understand why they are slow, and find out ways to optimize the user experience and overall ThingWorx server stability:

- ThingWorx Utilization Subsystem
- Thread-level information using Support Tools extension
- PSM PurePath analysis
- VisualVM

You can monitor the slow operations using any of the tools. You can use one or more tools together to understand the root cause of the slow operations in the application.

Using Utilization Subsystem to Monitor Slow Execution of Services

When you enable the Utilization Subsystem, it gets detailed metrics about the duration of service execution. The following details are retrieved using the services in this subsystem:

- Time a service takes to compete execution (minimum, maximum, and mean time)
- Number of times a service is executed

Statistics are recorded in milliseconds from the time that the server was last restarted, but can be persisted by enabling the **Enable Statistics Persistence** option. Use the `WriteStatisticsReport` service to create a CSV file that contains these statistics. The CSV file is created in the `ThingworxStorage` folder.

The metrics are collected only after a service is complete. If a service executes in an infinite loop that eventually brings the server down, the time for that service is not recorded.

It is recommended to export and analyze data from the Utilization Subsystem to identify long-running services. If you have identified a slow service, update the services to print logging lines. These logs can retrieve the duration of operations. Sometimes, a specific set of parameters or a specific query is responsible for the slow behavior. Logging the parameters in the script is an easy and reliable way to capture this data. The following script can be expanded to print detailed information about the operations that take more than 30 seconds. The script can be further updated to print service parameters or other data as required:

```
var startTime= new Date();
// potentially slow operation
var endTime = new Date();

//calculate difference in seconds and print details only if
operation takes over 30 seconds
var diff = (endTime.getTime() - startTime.getTime) / 1000 % 60;
if (diff > 30) logger.error("+++Slow operation: " + diff +
"seconds "); // add any parameters or queries
```

Monitoring the ThingWorx Subsystems

You should regularly monitor the performance of ThingWorx subsystems in your environment. The status of all subsystems is available in Composer in the **Monitoring** menu.

It is recommended to monitor the following subsystems:

- Event Processing Subsystem
  - This subsystem executes the event, timer, and scheduler logic in your ThingWorx application.
  - If the number of `activeThreads` is the same as the `corePoolSize` and the queue is accumulating events, it is possible that one or more services are either slow to execute or are blocking the subsystem.
  - Slow performance of the subsystem can degrade the performance of the entire system. You must use the monitoring tools to analyze the problem.
- Stream Processing Subsystem and Value Stream Processing Subsystem
  - These subsystems are used for ingesting data from your remote devices or business logic.
  - If the `queueSize` in the monitoring section is increasing constantly, it indicates that the system is not able to ingest the data. You must use the monitoring tools to analyze the problem.
- WebSocket Execution Processing Subsystem
  - The subsystem processes remote device logic for both incoming and outgoing requests.

- If you see many active threads and an increasing queue size, it indicates a throughput issue at the device communication layer. These issues should be investigated.

Using Thread Data to Monitor Slow Execution of Services

Slow services are identified and debugged by checking the thread-level data captured over a series of snapshots. This section describes the common thread issues. It also explains how the thread operations are related to services. These services are executed either by users or by your ThingWorx application using schedulers, timers, or other events.

For thread-level analysis, you should capture several snapshots. The snapshots show how a thread activity changes over time. In case of slow performance, generally, five to ten thread captures for a period of 5 minutes helps to identify the performance issues.

The Support Tool utility captures thread snapshots every 30 seconds. To enable this functionality, use the `CreateWatchdogFile` service or manually create a file called `runstacktrace` in the `Support.Tools` repository on the disk. After taking the thread snapshots, delete the watchdog file by running the `DeleteWatchdogFile` service or remove the `runstacktrace` file manually. It you keep the functionality running, several GB of thread data accumulates every few minutes causing disk issues.

Reading the Thread Dumps

Each thread dump starts with a timestamp when the snapshot is collected:

```
Full thread dump taken at Thu Aug 1 07:19:59 EDT 2019
```

JVM thread dumps show the call stack involved in any server operations running on a specific thread. A call stack does not indicate an error on the server, though a frequent error also displays a call stack. A call stack shows in reverse order how specific methods were invoked. See the following example:

```
"http-nio-8080-exec-8" tid=0x114 in RUNNABLE
Blocked: 32[-1ms], Waited: 99[-1ms]
User CPU: 109ms
    - synchronizer <0x4206a205> (a java.util.concurrent.
ThreadPoolExecutor$Worker)
    at sun.management.ThreadImpl.dumpThreads0(Native Method)
    at sun.management.ThreadImpl.dumpAllThreads(Unknown Source)
    at com.thingworx.support.utilities.Stacktrace.dumpThreadInfo
(Stacktrace.java:70)
    at com.thingworx.support.utilities.Stacktrace.dumpAllStacks
(Stacktrace.java:64)
    - locked <0x4cdd9461> (a java.lang.Class)
    at com.thingworx.support.utilities.Stacktrace.dumpAllStacks
(Stacktrace.java:38)
    - locked <0x4cdd9461> (a java.lang.Class)
    at com.thingworx.support.utilities.SupportToolsTemplate.
DumpAllThreads(SupportToolsTemplate.java:202)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown
Source)
    at java.lang.reflect.Method.invoke(Unknown Source)
    at com.thingworx.common.processors.ReflectionProcessor.
processService(ReflectionProcessor.java:261)
    at com.thingworx.handlers.ReflectionServiceHandler.
processService(ReflectionServiceHandler.java:50)
    at com.thingworx.things.Thing.processServiceRequestDirect
(Thing.java:7886)
    at com.thingworx.things.Thing.processAPIServiceRequest(Thing.
java:7824)
    at com.thingworx.webservices.BaseService.handleInvoke
(BaseService.java:3077)
    at com.thingworx.webservices.BaseService.service(BaseService.
java:356)
    at javax.servlet.http.HttpServlet.service(HttpServlet.
java:742)
…
```

In the call stack, the Support Tools extension methods are used to get thread information. The `dumpAllStacks` method invokes the `dumpThreadInfo` method that requests the thread information from the built-in Java platform.

The name of the service invoking these Java classes is generally listed below the call stack. In this case, the `DumpAllThreads` service is listed as part of the call stack. The `processService` method always invokes the custom services. Checking for services that are listed after `processService` in the call stack, enables you to get the thread on which a specific service is executed.

A thread has the following states:

- RUNNABLE—Good state. It indicates that the thread has all the resources it needs to perform the operations.
- BLOCKED—It indicates that the thread is waiting for an object in another thread before it can complete the operation.
- TIME_WAITING—It indicates that the thread is either waiting for a synchronizer to access a resource, or is waiting for some additional input (similar to WAITING state)
- WAITING—It indicates that threads are not doing any work. It is waiting for further input. This is the default normal state for 99% of the threads in a capture.

Note

If you collect the output using the Java `jstack` command instead of the Support Tools extension, many threads are shown in BLOCKED state. However, the threads are in TIME_WAITING state. This may be the default state for some types of threads.

Types of Threads in ThingWorx

The following threads are generally available by default in the ThingWorx environment. Depending on your database or customizations, you may see some additional thread types.

- `https nio` (Number of threads 1-1024+)

  Threads that are used for user requests or REST payloads. All external requests initially come through the `nio` (non-blocking IO) threads before executing on a different thread type. You can configure more than 1024 threads. In Apache Tomcat, the default number of threads is 150.
- `WSExecutionProcessor` (Number of threads 1000+)

  Threads that are used for remote device communications, for example, Kepware. Generally, only a few of these threads are actively running; 99% of the time these threads are in waiting state. These threads are configured using the WebSocket Execution Processing Subsystem. A bottleneck in the device communication is identified if all these threads are busy, or if the WebSocket Execution Processing Subsystem shows a growing queue of events.
- `TWEventProcessor` (Number of threads 16+)

  Threads that are used to execute code for all events, timers, and schedulers. When these threads are busy, events go in a queue. If the queue backs up significantly, it may impact the server performance. If all the event threads are constantly busy, the server may start to become unresponsive and may fail to respond to user input. These threads are one of the main thread types, and it is recommended to monitor them. The number of active threads is configurable and can be monitored using the Event Processing Subsystem. A bottleneck in event, timer, or scheduler execution can be identified if all the threads of this type are busy. In this case, `activeThreads` in the Event Processor Subsystem is consistently high and there is a growing queue of events in this subsystem in the ThingWorx Composer.
- `C3P0` (Number of threads 3-8)

  Threads that manage all the database connections. If the threads are blocked for long periods, it leads to database access issues.

- Timer or Schedulers (Number of threads 30-40)

  Threads that are used to monitor and trigger schedulers and timers. The code execution occurs on `TWEventProcessor` threads. These threads do not perform significant processing work. They time the events that are required to start additional processing.
- `pool` (Number of threads 40+)

  Threads that are used for writing persisted properties, stream, and value stream entries. An internal timer triggers these threads and most of the time the threads are idle. Some of these threads correspond to the Stream and Value Stream Processing Subsystems. If the queue in these subsystems increases, it indicates database throughput issues. The corresponding threads are saturated with activity.
- Async threads (Number of threads 1-100+)

  Business logic is triggered on separate threads if the underlying services are marked as async. The number of async threads can spike as high as several thousand. The name of the service that triggers the thread is a part of the thread name.
- Various JVM or Tomcat Apache threads (number of threads: 40+)

  Threads that are used for internal server operations at the Tomcat Apache layer. Generally, these threads are not analyzed as they do not trigger performance bottlenecks.

Issues with Threads

The following are some of the scenarios that indicate performance or other stability issues in your ThingWorx application:

- Many blocked threads for a long period of time
- Shared resource contention
- Long running threads

This section provides overview of these types of issues.

Blocked Threads

Many blocked threads for a long period of time indicate high resource contention at the JVM level or a potential deadlock. When numerous threads are blocked, and are waiting for a shared resource, for example, database access, users are generally blocked from performing operations.

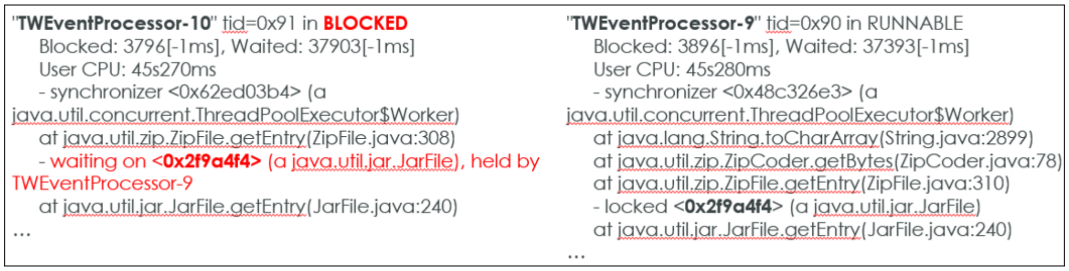
High contention sometimes resolves itself. For example, when a large database write operation is complete. However, if the threads remain blocked for a long period of time:

- There may be a deadlock situation, that is, threads lock each other in a circular pattern.
- Or, there may be a single blocking transaction, which does not complete processing in the required time

You can check for this issue by:

- Searching for number of blocked threads.
- Check if the thread that is blocked in one snapshot remains blocked in the subsequent snapshots.

The following image shows a blocked thread:



```
"TWEventProcessor-10" tid=0x91 in BLOCKED
    Blocked: 3796[-1ms], Waited: 37903[-1ms]
    User CPU: 45s270ms
    - synchronizer <0x62ed03b4> (a
java.util.concurrent.ThreadPoolExecutor$Worker)
    at java.util.zip.ZipFile.getEntry(ZipFile.java:308)
    - waiting on <0x2f9a4f4> (a java.util.jar.JarFile), held by
TWEventProcessor-9
    at java.util.jar.JarFile.getEntry(JarFile.java:240)
...
```

```
"TWEventProcessor-9" tid=0x90 in RUNNABLE
    Blocked: 3896[-1ms], Waited: 37393[-1ms]
    User CPU: 45s280ms
    - synchronizer <0x48c326e3> (a
java.util.concurrent.ThreadPoolExecutor$Worker)
    at java.lang.String.toCharArray(String.java:2899)
    at java.util.zip.ZipCoder.getBytes(ZipCoder.java:78)
    at java.util.zip.ZipFile.getEntry(ZipFile.java:310)
    - locked <0x2f9a4f4> (a java.util.jar.JarFile)
    at java.util.jar.JarFile.getEntry(JarFile.java:240)
...
```

In this example, the unzip operation in `TWEventProcessor-10` is blocked. It is waiting for another unzip operation to complete in the `TWEventProcessor-9` thread. Two users could be unzipping the files at the same time in ThingWorx, and the second user has wait for the first one to finish the operation. Both the operations are accessing object `0x2f9a4f4` which is locked in `TWEventProcessor-9`.

If many event processor threads are also blocked, there can be bigger server stability issues. Additionally, the Event Processing Subsystem may show an increase in the number of active threads. It also shows a growing queue of unprocessed events if the blocked scenario does not resolve quickly.

Shared Resource Contention

The database and event threads in ThingWorx can be a performance bottleneck for the following reasons:

- Check the contention for `C3P0PooledConnectionPoolManager` and `TWEventProcessor` group of threads. A large number of blocked threads or threads actively working in these categories may indicate a potential performance bottleneck.
- There may be events queued up on the server if all the database and event threads are occupied.

Consider the following example. In this case, an event thread blocks all the database connector threads. The event thread 11 in the call stack is performing a large database update operation.

```
"C3P0PooledConnectionPoolManager[identityToken-
>2v1py89n68drc71w36rx7|68cba7db]-HelperThread-#7" tid=0xa0 in
BLOCKED
    Blocked: 414717[-1ms], Waited: 416585[-1ms]
    User CPU: 5s320ms
     at java.lang.Object.wait(Native Method)
     - waiting on <0x56978cbf> (a com.mchange.v2.async.
ThreadPoolAsynchronousRunner), held by TWEventProcessor-11

"C3P0PooledConnectionPoolManager[identityToken-
>2v1py89n68drc71w36rx7|68cba7db]-HelperThread-#6" tid=0x9f in
BLOCKED
    Blocked: 415130[-1ms], Waited: 416954[-1ms]
    User CPU: 5s530ms
     at java.lang.Object.wait(Native Method)
     - waiting on <0x56978cbf> (a com.mchange.v2.async.
ThreadPoolAsynchronousRunner), held by TWEventProcessor-11
```

The logic of the custom services performing the operation can be optimized as shown in the following example:

```
"TWEventProcessor-11" tid=0x92 in RUNNABLE
    Blocked: 3913[-1ms], Waited: 37924[-1ms]
    User CPU: 45s430ms
     at com.thingworx.persistence.common.sql.SqlDataTableProvider.
updateEntry(SqlDataTableProvider.java:13)
     at com.thingworx.persistence.TransactionFactory.
createDataTransactionAndReturn(TransactionFactory.java:155)
     at com.thingworx.persistence.common.BaseEngine.
createTransactionAndReturn(BaseEngine.java:176)
```

Long Running Threads

Long-running threads indicate that the custom code in your ThingWorx application may need optimization. Typical user transactions complete quickly at the JVM level, in subsecond or a few seconds range. However, a long-running thread, for example, running for 10+ minutes, involving a custom code can indicate a problem. Gather several thread snapshots over a 10-minute interval to see how long an operation takes to execute on a specific thread.

For example, a capture during a 60-minute interval shows the same execution pattern on the same thread:

```
http-nio-8443-exec-25" tid=0xc7 in RUNNABLE
     Blocked: 750[-1ms], Waited: 8866[-1ms]
     User CPU: 1h35m
     - synchronizer <0x35153c2f> (a java.util.concurrent.
ThreadPoolExecutor$Worker)
     at com.thingworx.types.data.sorters.GenericSorter.compare
(GenericSorter.java:93)
```

```
     at com.thingworx.types.data.sorters.GenericSorter.compare
(GenericSorter.java:20)
     at java.util.TimSort.countRunAndMakeAscending(TimSort.
java:360)
     at java.util.TimSort.sort(TimSort.java:234)
     at java.util.Arrays.sort(Arrays.java:1512)
     at java.util.ArrayList.sort(ArrayList.java:1454)
     at java.util.Collections.sort(Collections.java:175)
     at com.thingworx.types.InfoTable.quickSort(InfoTable.
java:722)
…
```

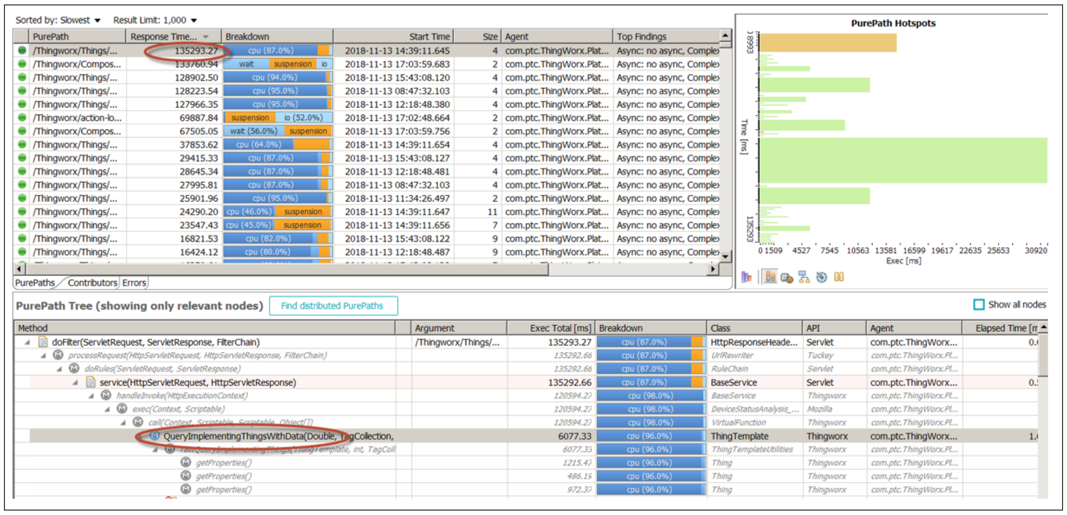
In the example, the thread is performing a sort on an infotable. The sorting operation continues while iterating through each item in a large infotable. The operation can be optimized to occur once at the end. Long running operations create contention either at the memory level, the database level, or on other server resources. Therefore, identifying and addressing long-running services in your custom ThingWorx application is important.

Analyzing PSM PurePaths to Monitor Slow Execution of Services

PSM records service execution metrics for some select operations over time. By default, PSM captures the data on http threads. Therefore, any bottlenecks in the Event Processing Subsystem or other ThingWorx subsystems are not collected in this tool. However, slow user operations are visible, and you can use PurePaths to determine which services and user actions are long running and should be optimized.

The Dynatrace PurePath technology records all user transactions and interactions performed on the ThingWorx server. PurePath records thread-level details that show how long each method took to execute. Additionally, PurePaths also record the duration for executing internal methods and shows how many resources each internal method consumed. PurePath also breaks down the wait time based on where the wait occurs, for example, slow IO, CPU, JVM suspension, and so on.

PurePaths give a direct view in the slow user transactions over time. For example, the following default PurePath dashboard enables you to sort services on the total execution time:



PurePath	Response Time...	Breakdown	Start Time	Size	Agent	Top Findings
/Thingworx/Things/...	135293.27	cpu (87.0%)	2018-11-13 14:39:11.645	4	com.ptc.ThingWorx.Plat...	Async: no async, Complex
/Thingworx/Compos...	133760.94	wait suspension io	2018-11-13 17:03:59.683	2	com.ptc.ThingWorx.Plat...	Async: no async, Complex
/Thingworx/Things/...	128902.50	cpu (94.0%)	2018-11-13 15:43:08.120	4	com.ptc.ThingWorx.Plat...	Async: no async, Complex
/Thingworx/Things/...	128223.54	cpu (95.0%)	2018-11-13 08:47:32.103	4	com.ptc.ThingWorx.Plat...	Async: no async, Complex
/Thingworx/Things/...	127966.35	cpu (95.0%)	2018-11-13 12:18:48.380	4	com.ptc.ThingWorx.Plat...	Async: no async, Complex
/Thingworx/action-lo...	69887.84	suspension io (52.0%)	2018-11-13 17:02:48.664	2	com.ptc.ThingWorx.Plat...	Async: no async, Complex
/Thingworx/Compos...	67505.05	wait (56.0%) suspension	2018-11-13 17:03:59.756	2	com.ptc.ThingWorx.Plat...	Async: no async, Complex
/Thingworx/Things/...	37853.62	cpu (64.0%)	2018-11-13 14:39:11.654	4	com.ptc.ThingWorx.Plat...	Async: no async, Complex
/Thingworx/Things/...	29415.33	cpu (87.0%)	2018-11-13 15:43:08.127	4	com.ptc.ThingWorx.Plat...	Async: no async, Complex
/Thingworx/Things/...	28645.34	cpu (87.0%)	2018-11-13 12:18:48.481	4	com.ptc.ThingWorx.Plat...	Async: no async, Complex
/Thingworx/Things/...	27995.81	cpu (87.0%)	2018-11-13 08:47:32.103	4	com.ptc.ThingWorx.Plat...	Async: no async, Complex
/Thingworx/Things/...	25901.96	cpu (95.0%)	2018-11-13 11:34:26.497	2	com.ptc.ThingWorx.Plat...	Async: no async, Complex
/Thingworx/Things/...	24290.20	cpu (46.0%) suspension	2018-11-13 14:39:11.647	11	com.ptc.ThingWorx.Plat...	Async: no async, Complex
/Thingworx/Things/...	23547.43	cpu (45.0%) suspension	2018-11-13 14:39:11.656	7	com.ptc.ThingWorx.Plat...	Async: no async, Complex
/Thingworx/Things/...	16821.53	cpu (82.0%)	2018-11-13 15:43:08.122	9	com.ptc.ThingWorx.Plat...	Async: no async, Complex
/Thingworx/Things/...	16424.12	cpu (80.0%)	2018-11-13 12:18:48.487	9	com.ptc.ThingWorx.Plat...	Async: no async, Complex

PurePath Tree (showing only relevant nodes)

Method	Argument	Exec Total [ms]	Breakdown	Class	API	Agent	Elapsed Time [m
doFilter(ServletRequest, ServletResponse, FilterChain)	/Thingworx/Things/...	135293.27	cpu (87.0%)	HttpResponseHeade...	Servlet	com.ptc.ThingWorx...	0.(
processRequest(HttpServletRequest, HttpServletResponse, FilterChain)		135292.66	cpu (87.0%)	UrlRewriter	Tuckey	com.ptc.ThingWorx.Pl...	
doRules(ServletRequest, ServletResponse)		135292.66	cpu (87.0%)	RuleChain	Servlet	com.ptc.ThingWorx.Pl...	
service(HttpServletRequest, HttpServletResponse)		135292.66	cpu (87.0%)	BaseService	Servlet	com.ptc.ThingWorx...	0.!
handleInvoke(HttpExecutionContext)		120594.27	cpu (98.0%)	BaseService	Thingworx	com.ptc.ThingWorx.Pl...	
exec(Context, Scriptable)		120594.27	cpu (98.0%)	DeviceStatusAnalysis_...	Mozilla	com.ptc.ThingWorx.Pl...	
call(Context, Scriptable, Scriptable, Object[])		120594.27	cpu (98.0%)	VirtualFunction	Thingworx	com.ptc.ThingWorx.Pl...	
QueryImplementingThingsWithData(Double, TagCollection,		6077.33	cpu (96.0%)	ThingTemplate	Thingworx	com.ptc.ThingWorx...	1.(
rawQueryImplementingThings(ThingTemplate, int, TagColl		6077.33	cpu (96.0%)	ThingTemplateUtilities	Thingworx	com.ptc.ThingWorx.Pl...	
getProperties()		1215.47	cpu (96.0%)	Thing	Thingworx	com.ptc.ThingWorx.Pl...	
getProperties()		486.19	cpu (96.0%)	Thing	Thingworx	com.ptc.ThingWorx.Pl...	
getProperties()		972.37	cpu (96.0%)	Thing	Thingworx	com.ptc.ThingWorx.Pl...	

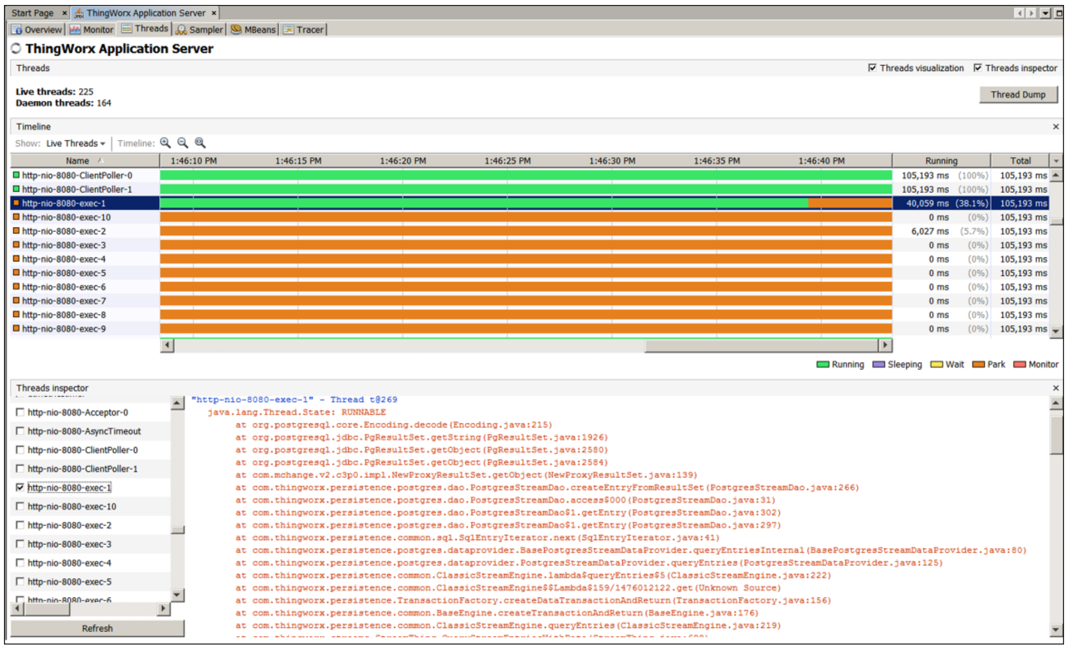
In the example, you can see that several services are performing slowly. These services impact the users. The users have to wait for over two minutes for a Mashup service to complete. PSM captures the end-user experience. It does not collect data for timers or schedulers that may impact backend processing.

Check the response time to identify services that need optimization. Dynatrace also helps you identify the longest API call in a service. In this example, the longest API call is `QueryImplementingThingsWithData`.

Using VisualVM to Monitor Slow Execution of Services

Use VisualVM to analyze threads that are in runnable state. The threads are visually indicated as a green line. If you see a line that remains green for an extended period of time, and this line is not one of the default JVM threads, then it is recommended to drill-down and analyze the transactions.

VisualVM does not record data over time. The performance issues should be analyzed and identified as they occur.



Live threads: 225
Daemon threads: 164

```
Name                           Running                 Total
http-nio-8080-ClientPoller-0   105,193 ms  (100%)      105,193 ms
http-nio-8080-ClientPoller-1   105,193 ms  (100%)      105,193 ms
http-nio-8080-exec-1            40,059 ms  (38.1%)     105,193 ms
http-nio-8080-exec-10                0 ms  (0%)        105,193 ms
http-nio-8080-exec-2             6,027 ms  (5.7%)      105,193 ms
http-nio-8080-exec-3                 0 ms  (0%)        105,193 ms
http-nio-8080-exec-4                 0 ms  (0%)        105,193 ms
http-nio-8080-exec-5                 0 ms  (0%)        105,193 ms
http-nio-8080-exec-6                 0 ms  (0%)        105,193 ms
http-nio-8080-exec-7                 0 ms  (0%)        105,193 ms
http-nio-8080-exec-8                 0 ms  (0%)        105,193 ms
http-nio-8080-exec-9                 0 ms  (0%)        105,193 ms
```

```
☐ http-nio-8080-Acceptor-0
☐ http-nio-8080-AsyncTimeout
☐ http-nio-8080-ClientPoller-0
☐ http-nio-8080-ClientPoller-1
☑ http-nio-8080-exec-1
☐ http-nio-8080-exec-10
☐ http-nio-8080-exec-2
☐ http-nio-8080-exec-3
☐ http-nio-8080-exec-4
☐ http-nio-8080-exec-5
☐ http-nio-8080-exec-6
```

```
"http-nio-8080-exec-1" - Thread t@269
   java.lang.Thread.State: RUNNABLE
        at org.postgresql.core.Encoding.decode(Encoding.java:215)
        at org.postgresql.jdbc.PgResultSet.getString(PgResultSet.java:1926)
        at org.postgresql.jdbc.PgResultSet.getObject(PgResultSet.java:2580)
        at org.postgresql.jdbc.PgResultSet.getObject(PgResultSet.java:2584)
        at com.mchange.v2.c3p0.impl.NewProxyResultSet.getObject(NewProxyResultSet.java:139)
        at com.thingworx.persistence.postgres.dao.PostgresStreamDao.createEntryFromResultSet(PostgresStreamDao.java:266)
        at com.thingworx.persistence.postgres.dao.PostgresStreamDao.access$000(PostgresStreamDao.java:31)
        at com.thingworx.persistence.postgres.dao.PostgresStreamDao$1.getEntry(PostgresStreamDao.java:302)
        at com.thingworx.persistence.postgres.dao.PostgresStreamDao$1.getEntry(PostgresStreamDao.java:297)
        at com.thingworx.persistence.common.sql.SqlEntryIterator.next(SqlEntryIterator.java:41)
        at com.thingworx.persistence.postgres.dataprovider.BasePostgresStreamDataProvider.queryEntriesInternal(BasePostgresStreamDataProvider.java:80)
        at com.thingworx.persistence.postgres.dataprovider.PostgresStreamDataProvider.queryEntries(PostgresStreamDataProvider.java:125)
        at com.thingworx.persistence.common.ClassicStreamEngine.lambda$queryEntries$5(ClassicStreamEngine.java:222)
        at com.thingworx.persistence.common.ClassicStreamEngine$$Lambda$159/1476012122.get(Unknown Source)
        at com.thingworx.persistence.TransactionFactory.createDataTransactionAndReturn(TransactionFactory.java:156)
        at com.thingworx.persistence.common.BaseEngine.createTransactionAndReturn(BaseEngine.java:176)
        at com.thingworx.persistence.common.ClassicStreamEngine.queryEntries(ClassicStreamEngine.java:219)
```

In the **Threads** tab, click **Threads Dump** to capture several thread dumps over time. This enables you to analyze the data offline.

In the example, the `http-nio-8080-exec-1` thread is active for an extended period. The corresponding call stack for the thread shows that the thread is executing a large query on the database to retrieve stream data. In this case, identify the service running on this thread and optimize it.

Best Practices for Improving Service Performance and Application Uptime

When you develop and monitor your ThingWorx applications, use the following best practices to ensure that your application remains responsive:

- Most services should complete under 10 minutes, even when they are executed on a timer or scheduler.
- Monitor ThingWorx subsystems daily to check for slow services.
- Implement a tool such as PSM to record the execution time of services.
- If there are services that perform complicated logic or database lookups, consider adding additional custom logging. The logs help you identify issues while troubleshooting slow performance issues.

Operating System and Database Performance Issues

The performance of your ThingWorx application depends on the performance of your underlying database engine. It is recommended that you closely monitor the health of your database engine. You should work with your DBA team to identify slow queries and address any latency or throughput issues between the ThingWorx application and the database. Generally, the expected latency between ThingWorx and the databases should be 10-15 ms. It is recommended that queries from your application to the database should complete in under a minute under normal load conditions.

Any slowdown in your ThingWorx application should be evaluated against database traffic, locked transactions, and slow queries. Monitoring tool such as PSM help in identifying the slow queries. This helps you to relate them to other events on the server. Additionally, database management tools provide a robust set of diagnostic tools to identify problem queries.

How to Monitor Operating System and Database Performance Issues

You can use the following tools to identify any database slowdowns that impact your ThingWorx application:

- PSM database transaction analysis
- VisualVM database pooling information
- Thread-level information collected using Support Tools
- ThingWorx and database log monitoring
- Database monitoring tools

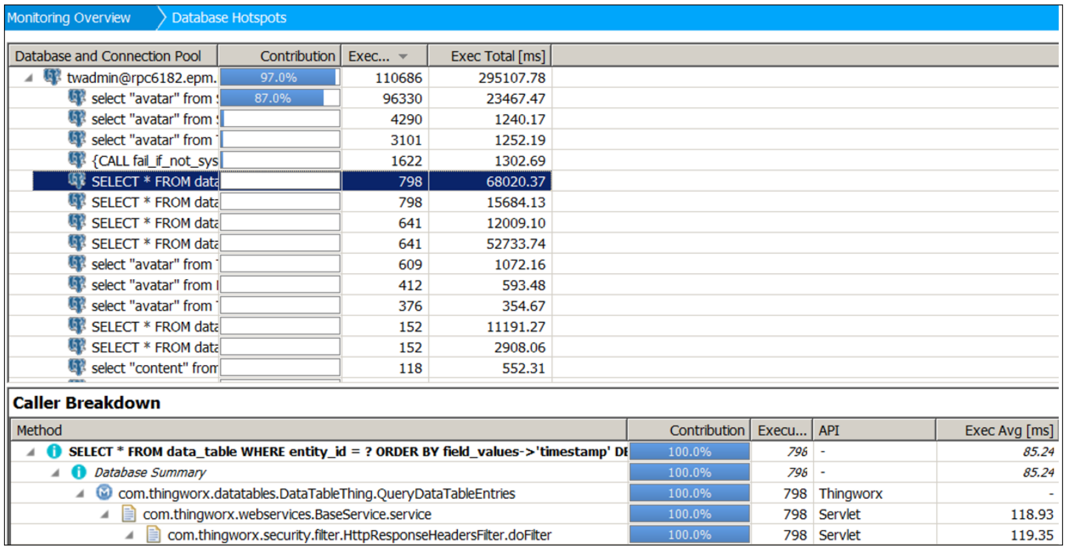
Using PSM to Monitor Database Performance

PSM captures most of the database queries initiated by your ThingWorx application. It has dashboards that highlight slow execution statements at the database level, connection pool usage, and captures the bind variables sent to the database.

PSM monitors transactions that are initiated by users. Therefore, PSM only tracks user-facing database transactions. If the database performance issue occurs in a scheduler, timer, or an asynchronous process, the corresponding queries are not collected by PSM.

For any period of slow performance, you can track the corresponding database transactions. Right-click any chart or dashboard to open the query view. For example, if you identify a CPU spike or slow PurePath, you can drill-down to the corresponding database queries occurring during that period.

In the following example, after you identify slow data table queries and their corresponding database queries, optimize the code in the application that performs the `QueryDataTableEntries` API:



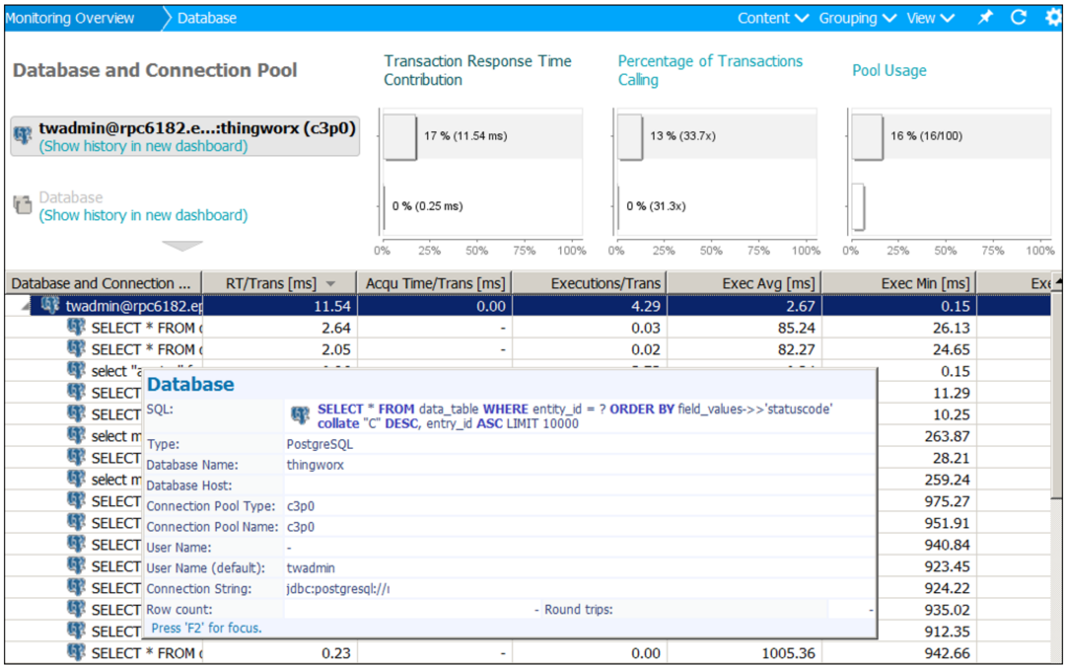
Monitoring Overview > Database Hotspots

Database and Connection Pool	Contribution	Exec...	Exec Total [ms]
twadmin@rpc6182.epm.	97.0%	110686	295107.78
select "avatar" from	87.0%	96330	23467.47
select "avatar" from		4290	1240.17
select "avatar" from		3101	1252.19
{CALL fail_if_not_sys		1622	1302.69
SELECT * FROM data		798	68020.37
SELECT * FROM data		798	15684.13
SELECT * FROM data		641	12009.10
SELECT * FROM data		641	52733.74
select "avatar" from		609	1072.16
select "avatar" from		412	593.48
select "avatar" from		376	354.67
SELECT * FROM data		152	11191.27
SELECT * FROM data		152	2908.06
select "content" from		118	552.31

Caller Breakdown

Method	Contribution	Execu...	API	Exec Avg [ms]
SELECT * FROM data_table WHERE entity_id = ? ORDER BY field_values->'timestamp' DE	100.0%	*796*	-	*85.24*
Database Summary	100.0%	*796*	-	*85.24*
com.thingworx.datatables.DataTableThing.QueryDataTableEntries	100.0%	798	Thingworx	-
com.thingworx.webservices.BaseService.service	100.0%	798	Servlet	118.93
com.thingworx.security.filter.HttpResponseHeadersFilter.doFilter	100.0%	798	Servlet	119.35

PMS displays the connection pooling details in your application. The database dashboard shows the overall usage of the connection pool as shown in the following example:



Database and Connection Pool

twadmin@rpc6182.e...:thingworx (c3p0)
(Show history in new dashboard)

Database
(Show history in new dashboard)

Database and Connection ...	RT/Trans [ms]	Acqu Time/Trans [ms]	Executions/Trans	Exec Avg [ms]	Exec Min [ms]	Exe
twadmin@rpc6182.ep	11.54	0.00	4.29	2.67	0.15	
SELECT * FROM d	2.64	-	0.03	85.24	26.13	
SELECT * FROM d	2.05	-	0.02	82.27	24.65	
select "a					0.15	
SELECT					11.29	
SELECT					10.25	
select m					263.87	
SELECT					28.21	
select m					259.24	
SELECT					975.27	
SELECT					951.91	
SELECT					940.84	
SELECT					923.45	
SELECT					924.22	
SELECT					935.02	
SELECT					912.35	
SELECT * FROM d	0.23	-	0.00	1005.36	942.66	

Database

SQL:	**SELECT** * **FROM** data_table **WHERE** entity_id = ? **ORDER BY** field_values->>'statuscode' **collate** "C" **DESC**, entry_id **ASC** LIMIT 10000
Type:	PostgreSQL
Database Name:	thingworx
Database Host:	
Connection Pool Type:	c3p0
Connection Pool Name:	c3p0
User Name:	-
User Name (default):	twadmin
Connection String:	jdbc:postgresql://ı
Row count:	- Round trips: -

Press 'F2' for focus.

ThingWorx uses a pool of connections. The default value for pool connections is set to 100. If all the connections in the pool are used simultaneously, the application may appear unresponsive. To resolve this problem, check if there are slow transactions holding a connection longer than necessary, before allocating additional connections.

ThingWorx uses the following connection pools:

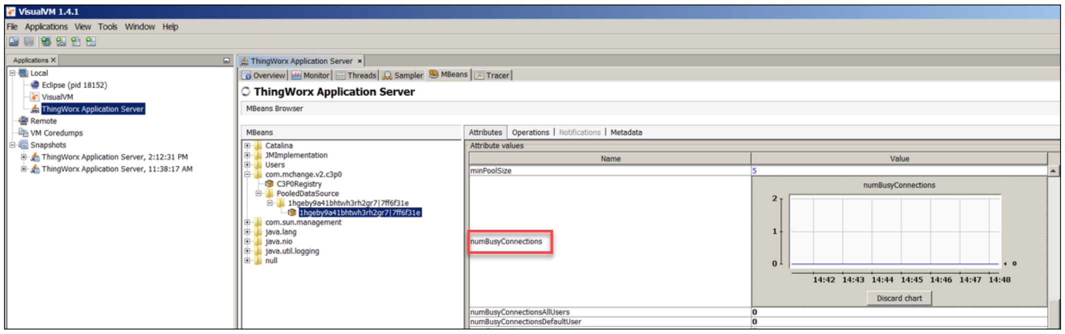
- Built-in persistence provider connections that are managed by the `C3P0` library.
- Other secondary JDBC connections, which are created from a database Thing, are tracked using Apache library.

In the PSM database overview dashboard, you can isolate periods of time when connection pool is saturated. You can check which queries are executing at the specified time. This helps you diagnose the connection pooling issues. You should resolve the underlying queries in your ThingWorx application that are holding a connection for longer duration of time.

Using VisualVM to Monitor Database Performance

VisualVM provides information about database transactions and connection pooling. To see connection pooling information, you should install the Java M-Beans plugin for your version of VisualVM. See the VisualVM documentation for more information on M-Beans plugin.

After you install the plugin, in the **MBeans** tab, you can identify the numbers of connections used in the connection pool. In the `c3p0` library, you can see the number of connections. It is shown as `numBusyConnections`.


VisualVM 1.4.1
File Applications View Tools Window Help
Applications
Local
Eclipse (pid 18152)
VisualVM
ThingWorx Application Server
Remote
VM Coredumps
Snapshots
ThingWorx Application Server, 2:12:31 PM
ThingWorx Application Server, 11:38:17 AM
ThingWorx Application Server
Overview Monitor Threads Sampler MBeans Tracer
ThingWorx Application Server
MBeans Browser
MBeans
Catalina
JMImplementation
Users
com.mchange.v2.c3p0
C3P0Registry
PooledDataSource
1hgeby9a41bhtwh3rh2gr7|7ff6f31e
1hgeby9a41bhtwh3rh2gr7|7ff6f31e
com.sun.management
java.lang
java.nio
java.util.logging
null
Attributes Operations Notifications Metadata
Attribute values
numBusyConnections
2
1
0
14:42 14:43 14:44 14:45 14:46 14:47 14:48
Discard chart

Name	Value
minPoolSize	5
numBusyConnections	
numBusyConnectionsAllUsers	0
numBusyConnectionsDefaultUser	0

VisualVM shows you which statements are running through this connection library. You can capture and analyze the thread dump to check which transactions are causing performance issues.

You can also use the Support Tools utility to capture thread dumps during periods of poor database performance.

In the thread dumps, check if a large number of threads are processing functions in the `com.thingworx.persistence.*` library. These threads should be isolated to identify the API calls to the database server.

Using ThingWorx Logs to Monitor Database Performance

ThingWorx logs should be monitored for any database issues. The following errors or warnings indicate a significant database connectivity issue and should be investigated:

Issue	Examples of log errors or warnings	Description
Apparent deadlock exceptions	`com.mchange.v2.async.ThreadPoolAsynchronousRunner$DeadlockDetector@635d735b` – APPARENT DEADLOCK!!! Creating emergency threads for unassigned pending tasks!	The error indicates that the connections in the connection pool are not returned quickly enough to the application. It rarely indicates an actual deadlock. Investigation—Isolate the slow-running SQL.
[2,006] unknown errors	Wrapped `java.lang.RuntimeException: com.thingworx.common.exceptions.DataAccessException`: [2,006] Unknown error occurred. Contact the administrator if this reoccurs.	The error indicates a basic connectivity exception when the database is suddenly disconnected. Investigation—Review the database and application logs to find the root cause of the disconnection issues.
[1,018] data store errors	[1,018] Data store unknown error: [Error occurred while accessing the model provider.]]	The error indicates that an individual statement generated an exception in the database engine. For example, while inserting a duplicate primary key in the table. Investigation—Review the database and application logs to find the root cause of the statement failure.

It is recommended that you review application logs daily for database errors. Analyze the errors or warnings. If you select the **Enable Stack Tracing** option in **LoggingSubsystem**, you can see the corresponding script or platform API that generated the failed database transactions. These failed transaction return relevant error codes.

Performance Monitoring Tools for the Database Layer

Most of the enterprise-grade database engines that can be deployed with ThingWorx provide good monitoring and issue detection tools. Work with your DBA team to monitor and analyze your database along with the application-side monitoring.

The following are some of the recommended issue detection queries for PostgreSQL and Microsoft SQL databases. These databases are the most used enterprise databases for ThingWorx model data.

Postgres

Monitor the following metrics to identify resource bottlenecks on the PostgreSQL server:

- Free disk space
- CPU usage
- I/O usage

You must identify and optimize the long running queries. Such queries can block other queries and increase wait time until resources are free again. This can result in long running threads waiting for a database lock on ThingWorx. A database lock enables an individual statement to lock the current state of row or table. Too many locks affect the performance of the application.

You can use PSM to detect long running queries as well as frequently executed queries.

MSSQL

For Microsoft SQL databases, it is recommended that you collect the database statistics on nightly basis in production environments. Additionally, indexes should be reviewed for high fragmentation and rebuilt as necessary, especially if the database is not running on SSD. The MSSQL Extended Events logs should be reviewed regularly for any long wait or deadlock operations in the ThingWorx database. During periods of slow performance, the following query can identify transactions that are waiting for a resource:

```
SELECT t.*,r.*, SUBSTRING(t.text, statement_start_offset/2 + 1,
(CASE WHEN statement_end_offset = -1
THEN LEN(CONVERT(nvarchar(max), t.text)) * 2
ELSE statement_end_offset
END - statement_start_offset) / 2)
FROM sys.dm_exec_requests AS r
CROSS APPLY sys.dm_exec_sql_text(r.sql_handle) AS t
```

If a statement appears suspended or is running for a longer duration, check the `wait_type` and `wait_resource` columns to identify the reason for the waiting time.

To see if the statement is waiting on any locks, identify the session ID and run:

```
select * from sys.dm_tran_locks where session_id = 158
```

Networking and Connectivity Issues

Your ThingWorx application performs and manages a significant number of transactions against external systems. The devices transmit large amounts of granular data using the network. This data is stored in a database or is processed by additional systems. ThingWorx may finally get this data from several external systems and display it in the Mashups in your application. This means that the performance of the network backing up all these components is critical for your application.

Any transmission or connection delay impacts the overall performance of your application. It is recommended that you monitor your entire network infrastructure for any transmission bottlenecks or other latency issues. Generally, the latency between frequently used components, for example, ThingWorx to database should not be more than 20 ms.

Network issues generally appear in your application in thread information. For example, the transactions that attempt to establish a connection or transmit data appear as slow. PSM, VisualVM, and Support Tools stack traces can identify the threads that are stuck performing network IO or connectivity tasks.

If a network issue is apparent in the thread-level or PSM PurePath details, it is recommended that you capture the network packets during periods of slow performance. Analyze the packet to identify the issues.

The network tools required to analyze network connectivity issues is out of the scope of this guide.

Additional Resources for Monitoring ThingWorx Applications

This section lists the additional resources that you can reference for a better understanding of the concepts.

Additional Resources for More In-Depth Learning

For more in-depth information on	Link	Resource
Installation Appendices Apache Tomcat Java Option Settings	http://support.ptc.com/help/thingworx_hc/thingworx_8_hc/en/index.html#page/ThingWorx/Help/Getting_Started/InstallingandUpgrading/Installation/apache_tomcat_java_option_settings.html	Help center
Monitoring Menu	http://support.ptc.com/help/thingworx_hc/thingworx_8_hc/en/index.html#page/ThingWorx/Help/Getting_Started/NavigationinThingWorx/MonitoringMenu.html	Help center
Logging Subsystem	http://support.ptc.com/help/thingworx_hc/thingworx_8_hc/en/index.html#page/ThingWorx/Help/Composer/System/Subsystems/LoggingSubsystem.html#wwID0EVO3U	Help center

Index