thingworx®

Developing Great IoT Solutions

A Best Practice Guide for Designing Scalable and Maintainable Applications

Document Version 2.0

Document Revision History

Revision Date	Version	Description of Change
August 1, 2019	1.0	First version
April 13, 2021	2.0	Reskinned document, updated to reflect ThingWorx version 9.1 with a more developed sample app provided.

Introduction

ThingWorx is an incredibly powerful and versatile tool for developing IoT applications. It handles the toughest of all, the back-end network building, like it's no big deal, making even the most complicated plans for interconnected devices seem simple. It allows for systematic creation of things on the Platform, all which can connect easily through provided Edge software and be maintained and monitored in real time. Designed around customization, the Platform can handle nearly any use case with ease.

That said, development in ThingWorx, as with any Platform, can be entirely more challenging and more delicate an endeavor than at first it seems. Writing any old application is fairly straight-forward; throw together some Javascript services, create your remote devices, drag-and-drop some widgets onto a **mashup**[1] to display the details to end users, and you've got a proof of concept (PoC). In theory, it's very simple! However, there are some things which must be considered early on in order to make that transition from PoC to Go Live as seamless and straight-forward as it needs to be.

As with many forms of application development, there are specific approaches to application design which lend well to developing **IoT applications**[2]. Considerations need to be made for the fact that hundreds of thousands of things will often need to be connected in the end (**scalability**[3]), and that even despite this, small tweaks to functionality should be possible when software improvements become available (**maintainability**[4]). An enterprise ready application is one which is both scalable and maintainable, and tested to ensure quality streaming from the Edge to the end user. This requires a little IoT expertise on the part of the developers, and a lot of quality documentation on IoT best practice.

This document was designed to provide that level of experience in the IoT world. The first section contains overviews on each of the technical categories required in the design of a maintainable and scalable application. It provides a high-level overview designed for the business folks and decision makers to ensure that they know where time is best spent when prioritizing the application requirements.

The second section provides a set of detailed examples. These step developers through the process of writing a maintainable application, helping them to learn the skills required to build their own applications from scratch later on. This section will get their heads in the right place when it comes to thinking like an IoT architect. The audience is expected to be beginner level with ThingWorx, but to have some experience developing web applications in the past. A familiarity with Javascript really helps. These examples ensure that ThingWorx developers know exactly what they can do, and how to best go about doing it.

The third section provides a reference guide, with links to best practice knowledgebase (**KCS**[5]) articles and helpful tips about the "do's" and "don'ts" of ThingWorx application development. Any developer who completes the examples and makes use of this reference guide should be well on their way to designing a maintainable and scalable IoT application. Furthermore, with this download find an entity file that contains an entire sample application. Appendix III has documentation on this sample application, which can be used as a guide for developing the guts of a custom application.

Overview of Design Areas

The fact that an application falls under the IoT umbrella most certainly *does* impact how components of the application need to be designed. This is because in a traditional application, users or customers access the website directly, and in ways which tend to be far less resource intensive, requesting less data at a time and updating existing data more slowly and at a non-constant rate.

Within an IoT application, this is simply not the case. **Edge devices**[6] are often pushing property updates to the Foundation server all at once, many hundreds (or even thousands) of things at a time. Likewise, the amount of data that is collected is massive, requiring careful consideration when it comes to the retrieval of this data for end user consumption. The more **nodes**[7] you have in a network (the larger the number of connected devices hooked up to the Foundation server), the harder it will be for the **controller**[8] of that network (the ThingWorx Foundation Server) to manage the exchange of data between these many nodes. For this reason, careful consideration must be given in a few key areas in order to ensure the **scalability**[3] and **maintainability**[4] of an IoT application. Details on each key area are provided below.

Scripting

ThingWorx has many tools to help in the development process. Under the "Snippets" tab, developers can find many services and **Javascript functions**[9] to help them accomplish their objectives. Some of these are services provided by built-in resources or subsystems, like the "Copy" method, a common service found on the FileTransferSubsystem, used to transfer files between **repositories**[10] or from the Edge to the Foundation server (and vice versa). Some of these are Javascript functions to help with development, and sometimes the snippets are just bits of code which help to provide the syntax required to utilize ThingWorx-specific functionality (for instance, how to create or increment through info tables).

It is always best to try to use a snippet instead of developing an algorithm from scratch (for example for data aggregation or interpolation). This is because the snippets have been developed by ThingWorx developers and are optimized for use in ThingWorx. Likewise, they will almost always be supported in newer versions. Occasionally there may still be major functionality changes, as some things become deprecated over time, but typically an alternative built-in function is provided in these cases, allowing for very easy maintenance of services over time when the **OOTB**[11] (out of the box) functions are used.

Under the "Entities" and "Me" tabs, developers can find all the services for specific things in ThingWorx. This also allows them to utilize many services which do not appear in the service listings on the things themselves, as well as eliminating the need to flip back and forth between things. Some services can only be called by other services or loaded into mashups, but they can't be run on their own. These same services would also appear in the **REST API**[12], under the ServiceDefinitions view.

There are a wide variety of provided services on the Platform, and while some appear to have the same function, sometimes the way in which they perform those functions can make a difference in the end performance of an application. For instance, queries tend to be more performant than searches when returning entities. If at any time you are concerned about which is the best way to design a certain service, never hesitate to seek help on the PTC Community or from the ThingWorx Technical Support team. It is always better to ask a question in advance than to run into performance issues way down the line. PTC Technical Support is happy to provide proactive help and participate actively in the planning process.

When determining the best way to write a service, it is important to explore all of the relevant snippets, and to consult the ThingWorx Best Practices Hub for more details.

When writing queries which obtain information for display on mashups, it is important to implement just the query itself first. Then, provide some search parameters and see how the performance looks. If the service takes a long time to return (say greater than 10 seconds), then perhaps consider an alternative approach. Also keep in mind that for some use cases, long queries like this will be unavoidable at times; in these cases, be sure to isolate the longer queries onto mashups of their own, so other things can be monitored in real time, and use good user communication (loading bars and animations) to reassure users that nothing has gone wrong on a webpage just because the queries take quite a long time to complete.

There are times where very long queries with complex processing logic are needed, particularly those which pull data from external sources, interpret it, and store it in Foundation data structures for use on mashups and throughout the application. For long queries which are expected to effect system performance, or if there are many, many smaller queries occurring at once and server slowness or unresponsiveness results, then consider making use of the external **Query Microservice**[25], functionality introduced in version 8.4.0 (the Help Center has overview information and installation and configuration details).

When building queries, consider using the "source" field as a way to query information more quickly, without having to apply a query after the full (and much larger) data set returns. For example, if data is segmented by fiscal quarters and searching for data by quarters is a major part of the use case, then consider assigning the quarter to the source field value or querying by data tag. Also, be sure to provide timestamps to all queries, not within the query structure itself, but directly into the parameters of the service (when available). This will greatly limit the amount of data returned from the database, to which the query is then applied.

Likewise, consider how much logic the query service must perform before returning. If it is something simple like modifying the syntax of one field (say from all caps to something prettier), then that is probably fine to do, even on mashup load. However, copying information from one **info table**[13] to another should almost never be done in a service meant for use on a mashup. Info tables are very memory intensive and utilizing them too often can cause system performance issues once real users get onto the mashups. As much as possible, avoid querying a data source in the same service where it is updated to ensure there are no **deadlocks**[14] or **race conditions**[15] affecting server stability.

Looping

Loops in an IoT application should be used sparsely and wisely. Anytime a long list of data needs to be analyzed line by line, the cost of that type of operation should be considered. Scrolling through an info table of even a couple hundred lines can affect performance and stability drastically, especially on mashup load. Poor use of loops can even affect the stability of the whole server, if the service is called too often or by many things at once. Loops should be used carefully in services and avoided wherever possible. However, they are often necessary, so it is important to bear in mind a few small tips.

One important thing to keep in mind is, pulling large chunks of data from the main data source repeatedly for use in different operations can add significant **cost**[16] to the overall load time of an application. Cost in this case refers to the length of time a service takes to complete, the runtime. Writing many services which pull the same data out of the database to process that data in different ways is very expensive. It is always better to loop through the data one time, doing whatever operations are necessary, pulling out data into smaller, more manageable data structures for further processing if needed, something often referred to as “caching” or “batching” data requests.

It is also important to avoid nested loops wherever possible. The cost of such operations is even greater and must be taken into account. This is sometimes unavoidable, but it is worth considering the design of the underlying data structure very carefully to try to avoid such requirements. For instance, avoid putting info tables as fields in **streams**[17] or **data tables**[18]. If the information in an info table needs to be stored for historical reference, consider giving it its own data table or stream instead. Alternatively, pull the relevant info out of the info table and store it in an external data structure.

As with all forms of scripting, testing small sections of code for performance is critically important. Additionally, **load testing**[19], especially on services with many loops, is strongly encouraged before any changes are considered finalized and ready for **Production**[20]. This means that many, many data points should be created for testing purposes, and it is important that the data resembles that which would be found in Production. This assures the scalability of the application, and therefore the stability of the server before anything has a chance to cause trouble for end users.

Data Reference

This is the trickiest, and yet most important component in building a scalable and maintainable IoT application. Choosing where to store data and how to reference it throughout the application is critical. Look to the end of this rather long and technical subsection to see a high-level overview comparing each structure. For a more detailed review of data structures and references, read on.

Memory in an IoT environment is very precious, and the cost of doing certain operations needs to be considered carefully when architecting the solution. There are several ways to store data in ThingWorx, in info tables, data tables, streams, value streams, or as properties on the things themselves.

Info tables are quick and easy to access and adjust, but they are very memory intensive. They sit in **JVM memory**[21] and eat up the resources needed to ensure the application runs smoothly. Info tables should almost never be used as **session variables**[22], especially if they are expected to have more than a hundred rows. If there are many users accessing mashups powered by info table session variables at a time, then not only will the mashups perform poorly, but the entire Foundation server may be at risk for instability and unexpected downtime. Likewise, info tables should not be written as a single field to a stream or value stream (or marked as "Logged"), as this too will take up a lot of memory and affect server stability.

This same concept will apply to info tables as properties on remote things. Info tables may be easy to update and access, but if there are thousands of remote things, each with several info tables of many hundreds of lines or more apiece, then the amount of memory required to sustain this environment will be very high. This design would not be very efficient and special consideration would have to be given to the amount of memory available to Foundation server. Instead, consider storing the data in one central location, like a stream, value stream, or data table. This has its own nuances, however, which will be discussed below.

Info tables are not really meant to keep track of historical data, instead being used for temporary information with limited numbers of rows. For example, it isn't a good idea to use an info table property to keep track of events happening on the edge devices. Who knows how many rows these will eventually have? It is almost certainly high enough that storing the data in a centralized location (like a stream) would be better. However, it would be perfectly fine to use an info table as a property to keep track of *which types* of devices exist on the platform. In this case, the number of options is limited, and the same info table can be referenced by many different resources. The info table with the types of devices can then be bound to a list widget on an asset creation screen and referenced in the scripts which create the assets.

For historical data specifically, consider streams and value streams. These are both very similar, the main difference being that one is updated automatically when properties are marked as "Logged" (value streams), and the other requiring manual updates or updates via subscription (streams). For both of these, data should go in and rarely be modified. These reflect time series data primarily, events, button clicks, property history information, etc.

However, their purposes differ in the grand scheme of the application design. Value streams are for short-term storage of data, i.e. every property value update which happens for a thing over a short period of time. Mechanisms on the Foundation server would then aggregate this data, converting it to fewer data points and writing it to other streams. Streams are primarily designed for more long-term storage of data, as places to keep track of say, the last 3 years' worth of daily revenue information. Using these data structures in this way isn't necessary, but it is highly encouraged in order to promote healthy maintenance of server data.

Having a purge mechanism becomes easy if this best practice is upheld. Then, all value stream data can be purged safely after a number of days, and stream data can be stored for many years. For instance, having 180 data points per hour for 2,000 things may be too much to store for more than a few days, as a week's worth of data gathered at this rate equates to roughly 60.5 million data points in a single table (which can affect lookup times, cause the database to run out of transactions, and backup the stream processors with requests). However, having 365 data points a year, even for tens of thousands of things, is still not too bad.

This is why aggregation is so helpful. If you find ways of viewing the data such that a single entry with several different values per day is enough (e.g. daily revenue, product energy usage for the day, etc.), then years' worth of data can safely be kept with minimal efforts required to maintain it. Use the value streams for data ingestion, and consider Influx DB (a persistence provider designed for time-series data, added in 8.4) if lots and lots of data comes in from many things at once. This data can be reviewed on mashups (though it is best to try and keep queries against ingested data sources to a minimum), but only query for small chunks of time at once, say for a display of several minutes of data on one thing at a time. However, for anything considered historical, consider smoothing the data, averaging several points together, and aggregating it down to much fewer points overall. Consider that viewing hours of data at a time only requires one data point per minute or per half minute, while viewing minutes worth of data at a time may need a data point per second, and these functionalities can pull from different sources on a mashup to ensure performance and stability.

Data tables are more robust, operating like info tables, but existing as their own entities on the Foundation server. These work more like conventional database tables (though not quite as robust), permitting for an additional field to be indexed (beyond what comes standard) for faster lookups. Data tables are row-locked, meaning that if a single row is updated at a time, the rest of the table is still accessible. This is another reason why queries perform faster on data tables, even while they are being updated by some other part of the application. Contrarily, stream-based queries often must wait for the updates to be completed, according to a queue-based system of updates and requests. Therefore, data tables are more efficient for data sources which are often manually modified by small numbers of end users or which are updated and reviewed independent of timestamps.

Data tables may require purge mechanisms as well, and so do info tables (depending on usage). Info tables should be limited to less than a couple hundred rows typically, sometimes even fewer than this, while data tables can have up to 100,000 rows. Streams can store many more rows than this (often in the 10s of millions), but remember, the more data there is to sort through, the slower the queries will return, and the worse the mashups will seem to perform to end users. This is why limiting the data which can be queried using date-time selector widgets should be considered a requirement of well-built applications.

An example use of data tables is provided in the second section of this document, where the concepts of aggregation and efficient querying are explored in depth. One final note about data tables is to be very careful where they are updated in the application; deadlocks are every server admin's worst nightmare, as the Foundation server slowly begins to crash and more and more data gets lost. This can occur when many things are trying to update the same data table rows at the same time. Try to build mashups so that if one person is modifying a data table manually, or if an automatic process is updating it, others cannot also update that table at the same time.

Finally, storing information on the remote things themselves is a good practice. Using the built-in GetProperties service allows for real-time updates on an individual remote asset level, so that when a device in a grid is selected, a display can show what its properties are, including meta properties (like the OS of the Edge device software, ID numbers, or device ownership information) as well as any remote properties which are frequently updated by the Edge (like device energy usage or device location, for example).

See the below chart for an overview of the different data structures.

In Standard SCP (Smart and Connected Product) Solutions

Data tables are for populating grids or trees on mashups. Data may often: • Update based on property changes on remote devices and/or • Be updated by small numbers of end users	**Streams** are for populating charts or querying data by timestamp. They store aggregated time series data for years at a time, and should make use of an automatic purge once each day (during "off hours").	**Value streams** are for ingesting data.They store raw data straight from the edge devices, or detailed property info, for days or weeks at a time, requiring an automatic purge.	**Infotables** are for populating menus and facilitating service logic. They hold temporary information, and should never get too large.	**Properties on remote things** are for instant reference (either on mashups or in other services), and for triggering data change events and alerts.
Database tables for permanent data, any which users update	*What you would look at from a distance, long-standing, historical data*	*Data ingestion which needs frequent purges, avoid use on mashups*	*Costly & inefficient, but easy to use (for impermanent data)*	*Reliable instant updates, but a bit memory intensive*

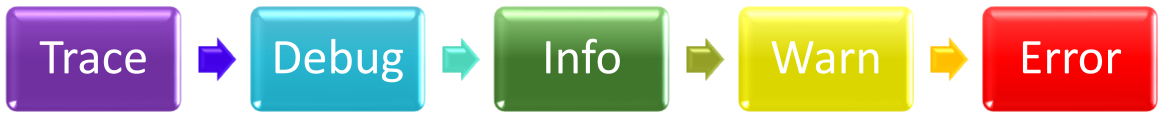
Less → More Transient Storage of Data

Logging

Logging is incredibly important to the maintenance of an application. When things go wrong, and things inevitably will go wrong, it is crucially important that technical support engineers can quickly and easily diagnose the problem and restore the system. The better the logging, the lesser the downtime. For many customers, downtime means a literal loss in revenue, so it is quite important to consider.

The Foundation server allows you to adjust what level of logging is written to the logs. Not enough logging, and technicians will not be able to diagnose the problem quickly. Too much logging, and the server memory will be bloated with log output, resulting in performance issues, CPU alerts, and potentially even server crashes, as log files take up all available memory and leave nothing for the rest of the application. So, the important take-away from this category is "everything in moderation", and here are some guidelines to help get that balance right.

Logging statements should be put at the start and end of just about every service, using a setting called "Trace". This level of logging is the highest, including messages which you don't normally see nor care to see. When all is well, and the server is functioning as expected, the log setting would normally be "Info" or even "Warn" level. The full order is:



Anything farther to the right of the setting in this image will be included, so "Trace" and "Debug" both are used to try to determine the root cause of issues. "Info" is used if something slightly out of the ordinary may or may not happen, and a server admin would like to know either way. "Warn" is for when something might be an issue, but it isn't clear, and "Error" is obvious: something has gone wrong.

In addition to the "Trace" statements at the start and end of services, "Debug" statements can be added when contained service calls are tricky, but not expected to fail (e.g. when adding a new remote thing), or to note that some part of a service is working correctly. These are the kinds of statements which log every time, no matter what happens within the service, and so they print many times to the logs, even while there are fewer places within the services themselves in which to place them (compared with errors, for instance).

"Info" can be used to note in the logs when a critical service completes successfully (like ones which update or aggregate data and prepare it for consumption via mashups), and should print similar information to the debug statements, but far less often, for instance inside of a check which determines if something was successful. "Warn" statements are commonly used throughout the various try and catch blocks, to denote if say an individual row of a table is not updating, even while most updates succeeded. "Error" will pop up in services all over the place, almost always within a try/catch block (unless "Warn" is used, as in the previous example), but ideally will log the least number of times (which means things are working as expected).

Logging statements should take a specific format to enable easier access to them. Have each statement start with something like "me.name: NameOfService:", which will allow technicians and admins to search by service. For example, if a technician wants to check that the 3 pm data update timer ran as expected, or how often some property change triggered some other update, etc. then they can just type the service or device name into the log sort mechanism, and see all related events from that service or device.

Also, be sure your developers always include the source or the original error message in the logging statement so that errors don't get "swallowed up" by your application. When that happens, it can double or triple the amount of time needed to diagnose the root cause! This means adding something like "Message: " + err.toString() to the logging function parameter (assuming the exception is called "err").

Ensure that there are clear and concise logging statements throughout the whole application, all assigned the appropriate logging level, and all will be well. A technician or admin should be able to glance at the logs to know exactly what is going on in the system. Runaway errors should be tracked down and fixed, and the logs should not appear flooded with too many messages. Find the right balance, and application maintenance will be a breeze.

Error Handling

In any application, when an error happens, it is sent to the logs, sometimes re-transmitted back to the host site, and then, dismissed of importance. In an IoT application, the option exists to handle this error in a slightly smarter fashion, instead storing its instance in a stream or perhaps displaying the specifics of the error on a special dashboard designed for internal use only. This allows for those less knowledgeable about the inner workings of the app (business users) to look at and maintain the edge devices on their own, without always needing to consult more technical people for help.

Writing errors to the logs does help more technical administrators, and it certainly helps the Technical Support Engineers (TSEs) at PTC who are happy to assist with troubleshooting and diagnosing issues. However, it does nothing to assist the end user, and it requires that administrators have a certain level of technical knowledge to be able to identify issues. A better approach, especially for applications which are then resold to less experienced application owners, is to build screens within the application which certain admin level users can access to review which issues their users are having.

This might include information like: device downtime for devices which have disconnected an abnormal number of times, metrics of performance from devices which seem to be operating outside of acceptable standards, or issues within the Foundation application itself, like errors from attempts to add new things, perform invalid queries, or update data sources for populating certain mashups. Then, administrators can change configurations within their own processes, open informed support cases on their own, and train their employees to use the application better, all in accordance with whatever trends they see on their “error handling dashboard”.

Browser Tools and Post-Development Analysis

Browser Tools[23] is an application built-in to most browsers which provides for more detailed information about **network requests**[24], i.e. the service calls which populate your mashups. The different options here can measure query performance, aid with debugging issues, and enable more detailed analysis of application performance. A detailed technical guide is provided at the end of the first example.

Post-Development Analysis is important to ensuring the application transitions well from PoC (Proof of Concept) to Production. It includes all forms of testing: QA, load testing, beta, etc. QA testing is standard enough practice to almost not even mention, but it is important to ensure that all things do what they are supposed to do and update or refresh as expected. Likewise, beta testing ensures that your end customers know how to use the application, and that no documentation or use case gaps exist. Regression testing should be a part of every stage of development to ensure nothing is broken over the many months it takes to assemble the application.

Most important of all is load testing, which must be done before any application can be considered complete. Load testing means creating many, many data points, which are similar in every way to what would exist in Production, and seeing how the application would perform. It is also possible to simulate that many devices are connected, which is critical for verifying the scalability and enterprise readiness of data ingestion and processing.

Example 1: Creating and Displaying Thousands of Things

- **Teaches how to:**
  - **Create and delete thousands of things**
  - **Write queries in ThingWorx**
  - **Use collections and grids, textboxes and buttons**
  - **Create basic mashups and interfaces for displaying a lot of data**
- **Covers:**
  - **Setting up a data model**
  - **Scripting services to interact with the model**
  - **Querying data contained within the model**
  - **Troubleshooting performance issues on mashups**

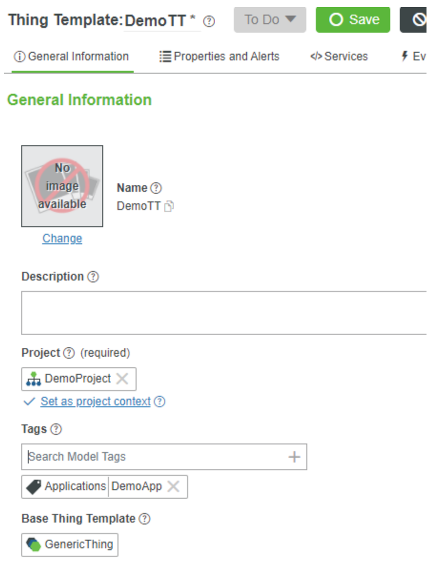
Context

This example completely breaks down the mechanism for thing creation and deletion, while also exploring how to access information from these things most efficiently. This example implementation is broken down into segments with notes and checkpoints in between each section to help the reader know when to take breaks and what to look for in each section. References throughout will link to Appendix I: Quick Tip Chart and Notes for additional details on the do's and don'ts of IoT development.

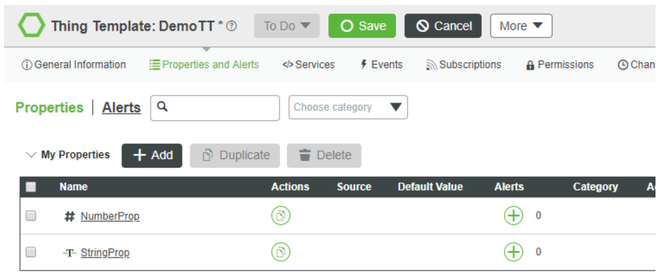
Note that any code snippets should NOT be copied directly from this document, as PDF rendering adds special characters that need to be removed before the service can be saved. Instead, take the time to type out each line, comments can be skipped, and use this as an exercise in learning the ThingWorx JavaScript syntax. Note that some are on two lines for display purposes and must be combined to one line in order to run correctly. There is built-in linting in ThingWorx to help identify these lines.

The purpose of this example is to exhibit some safe ways to build UIs which display thousands of things. The collection widget will be contrasted against the grid widget in terms of performance and capability, while the efficacy of different forms of queries and filters will be considered. Near the end, Browser Tools will be used to dig even deeper into query performance analysis.

Tutorial Section 1: Programmatically Create Many Things



1. Create a thing template called **DemoTT**
   a. Use the GenericThing parent template
   b. Add a tag as desired (here called DemoApp)
   c. Give the template two properties:
      i. NumberProp with type NUMBER
      ii. StringProp with type STRING



Name	Actions	Source	Default Value	Alerts	Category
NumberProp				0	
StringProp				0	

2. Create a GenericThing called **DemoAppUtility**
3. On **DemoAppUtility**, add a service called **CreateNThings**
   a. This should take an integer input called "N", which can be given a default value of 1000 for this demo, and a thing template input called "thingTemplateName" here
   b. This should return an integer
   c. In the code block, input the following:

```
var counter = 0;

for(var i = 0; i < N; i++) {
    var thingName = thingTemplateName + "_Device";

    try {
        // Each new thing needs a unique name, so either
        // pass it in from a UI, or generate it like this
        thingName = thingName + i;

        // Create the thing
        var params = {
            name: thingName,
            description: "Automatically created demo thing",
            thingTemplateName: thingTemplateName,
            tags: "Applications:DemoApp",
            projectName: "DemoProject"
        };
        Resources["EntityServices"].CreateThing(params);

        // Need this part because they do not get enabled
        // on their own when created programmatically
        Things[thingName].EnableThing();
        Things[thingName].RestartThing();
        // Update the thing with initial information
        // Any code can go here, e.g.setting initial property values on the new
things
        Things[thingName].NumberProp = Math.random()*100;
        Things[thingName].StringProp = "I've said Hello World " + i + " times.";

        counter++;
    } catch(err) {
  // Note that for formatting reasons, there is a newline and indentation added
  // here. This will need to be removed before running the service (just move
  // the second line to the same line as the first). There may be more modifications
  // like these required in all of the services if they are copy and pasted directly
        logger.error(me.name + ": Could not create thing named " + thingName
                  + "; Message: " + err);

        // As per KCS 198580, you MUST DELETE things created from runtime memory
        // straight away if they are not created successfully. Otherwise, you will
        // have ghost entities on the Platform //
        Resources["EntityServices"].DeleteThing({name: thingName});
    }
}

var result = counter;
```

   d. Run this service after completing it with N = 1000

4. On **DemoAppUtility**, add a service called **DeleteNThings**
   a. This should take a number input called "N", which can be given a default value of 1000 for this demo
   b. This should return an integer
   c. In the code block, put the following:

```
var counter = 0;

for(var i = (N-1); i >= 0; i--) {
    var thingName = thingTemplateName + "_Device";

    try {
        thingName = thingName + i;
        Resources["EntityServices"].DeleteThing({name: thingName});
        counter++;
    } catch(err) {
        logger.error(me.name + ": Could not delete thing named " + thingName
+ "; Message: " + err);
    }
}

var result = counter;
```

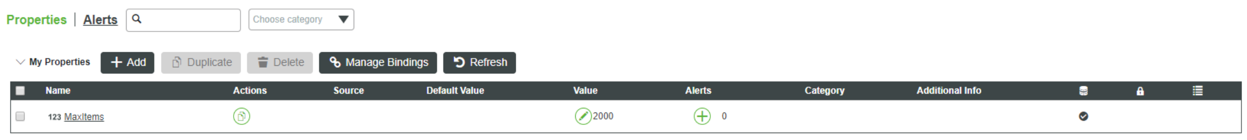
   d. You don't need to run this now; we need this service for later. If you ran it as a test, recreate the things using the **CreateNThings** service before going into the next section

End Section 1: Review

In this section, we learned how to create and delete things programmatically. Note how the services EnableThing and RestartThing are called after thing creation. This is done to avoid ghost entities, or entities which exist in JVM memory, but not in the database. Likewise note that the DeleteThing service call is in a try-catch block. This is extremely important because if the service deleting the things does not complete successfully, then none of the changes made on the JVM are pushed to the database. This results in the creation of reverse ghost entities, or those which exist in the database, but not on the Foundation server. Both this and the above issue can be resolved by an extension, in the case that issues do happen. See the appendix for further details.

Tutorial Section 2: Query Things Using Collection Widget

1. Create an INTEGER property (marked as persistent) on **DemoAppUtility** called **MaxItems**; set it to 2000



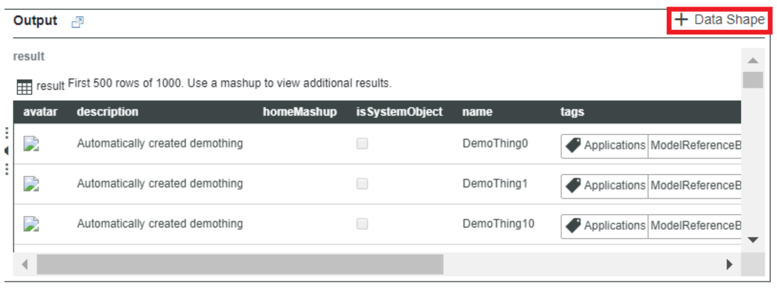
	Name	Actions	Source	Default Value	Value	Alerts	Category	Additional Info			
☐	123 MaxItems				2000	0			✓		

2. Create a service called **QueryDemoThings** on the **DemoAppUtility** thing

   a. There is one required input parameter for the thing template name, same as before

   b. In the code block, put:

```
var params = {
    // This pulls from a property so as to not be a hard-coded max in each service
    // This way when more things are eventually added, this number can be increased
    // Leaving this blank will result in only 500 entries being returned
    maxItems: me.MaxItems,
    query: undefined /* QUERY */
};

var result =
ThingTemplates[thingTemplateName].QueryImplementingThingsWithData(params);
```



result

result First 500 rows of 1000. Use a mashup to view additional results.

avatar	description	homeMashup	isSystemObject	name	tags
	Automatically created demothing		☐	DemoThing0	Applications ModelReferenceB
	Automatically created demothing		☐	DemoThing1	Applications ModelReferenceB
	Automatically created demothing		☐	DemoThing10	Applications ModelReferenceB

   c. Leave the query undefined for now

   d. This should return an infotable, but create the datashape manually:

      i. Run the service after it is complete, but while it is open so the “Inputs” and “Output” boxes are visible beneath the service editor

      ii. Then click "+ Data Shape"

iii. Name and tag your new datashape (**DemoThingDS**)

iv. Click to edit **QueryDemoThings** and add the data shape to the return infotable

3. Create a mashup called **InnerDemoMashup**

a. This is not the same thing as the Collection Widget

i. This is the mashup which will be tied to the widget

ii. It will appear once for each thing returned by the data source bound to the widget

b. In the past, this mashup was made static in size, something which is deprecated as of ThingWorx 9.1+, so the size will instead be specified within each container and within the Collection Widget on the outer mashup (all coming up)

c. Add a Mashup Parameter

i. Click the arrow in the top right corner

ii. You may have to select the mashup in the workspace to see the right arrow

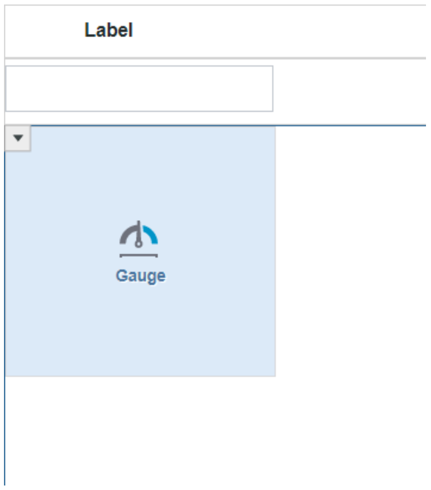
iii. Then click "Configure Mashup Parameters"

iv. Next, click "Add Parameter" (black button)

v. Call the mashup parameter "thingName"

vi. Give it the basetype THINGNAME

vii. Click "Done"



d. Configure the widgets

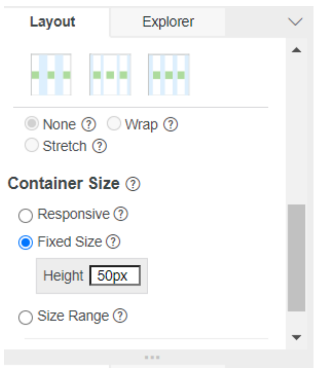
i. Put a label at the top of the mashup, but offset it, the thing name will populate and expand it to the right

ii. On the layout tab, add another container; use a fixed height for each

iii. Put a text field in this new container and set the width property to 200

iv. Add another layout with a gauge in it, also with a width of 200

v. Ensure the total mashup height is 275



None Wrap
Stretch

Container Size

Responsive
Fixed Size [✓]
Height 50px
Size Range

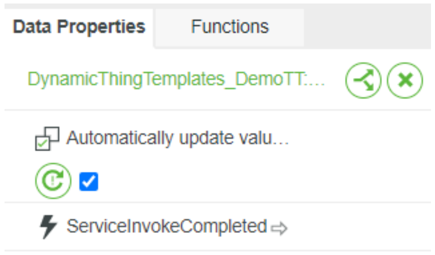
e. Add and Bind Data

i. In the data panel, click to import a new service

ii. Search for **DemoTT**, and be sure to check "Dynamic"

iii. Find the **GetProperties** service, and select it with the arrow icon; be sure to check the box labeled "Execute on Load" before you click "Done"

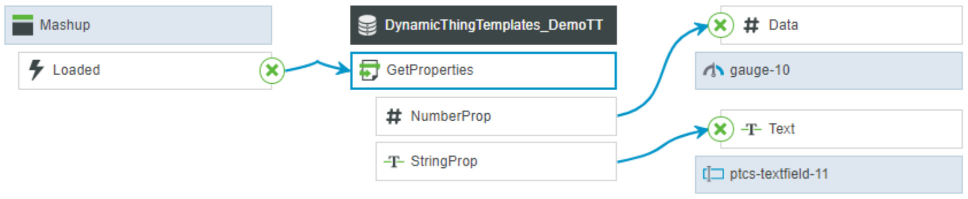
f. Highlight the **GetProperties** service in the data panel and check the box with the check mark icon labeled like "Automatically update values"


Data Properties
Functions
DynamicThingTemplates_DemoTT:...
Automatically update valu...
ServiceInvokeCompleted

i. Note that to get real-time property updates using the automatic update feature of **GetProperties**, the thing template name cannot be parameterized on this mashup. Instead, duplicate this mashup once for each thing template in the application (which usually correspond to different regions, if there are multiple regions on one ThingWorx instance)

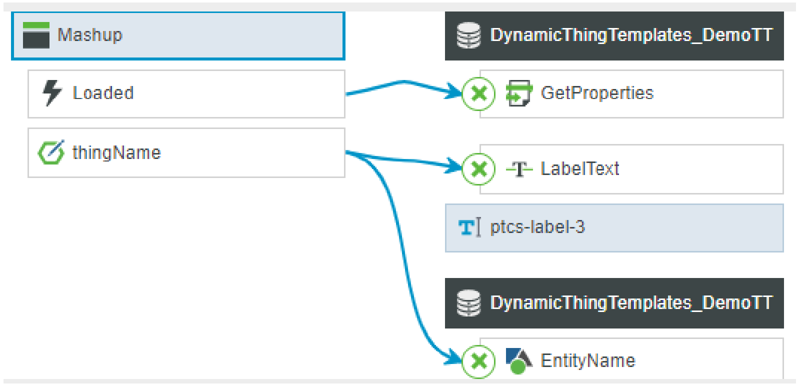
ii. This works best when there is a small number of end users; otherwise, consider alternative solutions to reduce the total number of internal websockets the Platform needs to render webpages)

g. Expand **GetProperties** and bind **StringProp** to the textbox (select "Text" in the popup) and **NumberProp** to the gauge (select "Data" in the popup)



h. Click in the workspace to select the mashup itself, and then bind the **thingName** mashup parameter (by highlighting over the arrow in the top-right corner) to the label widget (select "Text") and the **EntityName** parameter of **GetProperties**

i. Select the mashup in the workspace, and check that the full bindings should look like this:



j. Save and tag the mashup

4. Create another mashup called **DemoThingMashup**

a. Drag and drop a **Collection Widget** onto the mashup

b. Assign the Mashup property of the collection widget to **InnerDemoMashup** (found under **Properties** on the bottom left-hand side of the screen)

c. In the data panel, search for services on the **DemoAppUtility** thing

d. Select **QueryDemoThings** with the arrow icon

e. Check the box for " Execute on Load"

f. Click "Done"

g. Bind "All Data" to the Collection (select "Data")

h. Set the **MashupHeight** to 275 and **MashupWidth** to 200, then set the **UIDField** and **SortField** to "name"

    i. If the **UIDField** box doesn't populate, then the data shape was not added correctly in step 2c, part iv

i. Search for the **MashupPropertyBinding** property under the Collection Widget, and click "Add", then putting the following JSON in as the value: `{"name": "thingName"}`

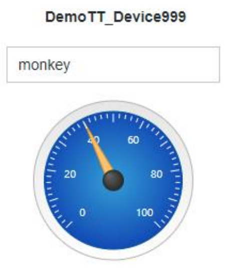
    i. Note that this has the property name from this mashup on the left side, and the name of the mashup parameter is on the right side

    ii. If this is backwards, then no data will be sent via the mashup parameter, so no label will appear, and no property information will populate, on the inner mashup

    iii. For now, hard code the **thingTemplateName** input property on this service to be "DemoTT".

    iv. If multiple thing templates are on the same server, add to this mashup some way for users to select which thing template they would like to review.

    v. This may mean selecting a region on another mashup which then opens this one (in the case where each thing template corresponds to a region, like in the Coffee Machine Example App), or it may mean selecting a type of device, etc.



j. Save and tag the mashup

k. View DemoThingMashup

    i. If nothing appears, there may be a permissions issue on the thing or thing template level, as they are not being returned (check logs for errors)

    ii. If the widgets appear but not data, see if step i. above was done correctly

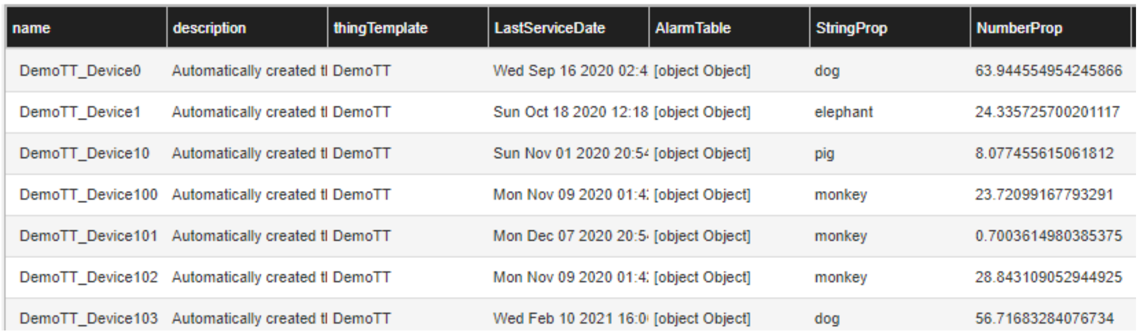
End Section 2: Review

In this section, we learned to create mashups and use several basic widgets, including the collection, text box, gauge, and label widgets. We necessarily explored passing mashup parameters from one mashup to another. There are a few best practice tips which demonstrate why using mashup parameters instead of session variables wherever possible is a good idea, see the appendix for details.

Many people use row-sized mashups in the collection widget to enable better rendering of charts and images inside of grids. Using static size containers on mashups is a good idea when creating custom grids in this way and when using mashups as pop-ups. However when possible, it is better to use the dynamic sizing for web pages so that they render well on every screen size (see the appendix for more details).

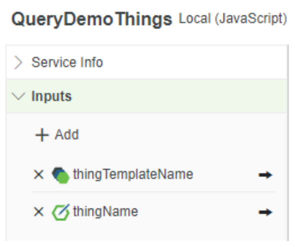
Tutorial Section 3: Query Things Using a Grid

1. Click to edit **DemoThingMashup** again
    a. Add some layout boxes, one on top with a fixed width of 100px (and static positioning if desired), and one to the right of the Collection Widget with an Advanced Grid in it
    b. Then, in the header container, add a text field and a button; more text fields and drop-down lists will be added to this container as the query becomes more complex
    c. Bind the QueryDemoThings "All Data" target to the grid, and select which columns to display
    d. Click Save and view the mashup



name	description	thingTemplate	LastServiceDate	AlarmTable	StringProp	NumberProp
DemoTT_Device0	Automatically created tl	DemoTT	Wed Sep 16 2020 02:4	[object Object]	dog	63.944554954245866
DemoTT_Device1	Automatically created tl	DemoTT	Sun Oct 18 2020 12:18	[object Object]	elephant	24.335725700201117
DemoTT_Device10	Automatically created tl	DemoTT	Sun Nov 01 2020 20:5	[object Object]	pig	8.077455615061812
DemoTT_Device100	Automatically created tl	DemoTT	Mon Nov 09 2020 01:4	[object Object]	monkey	23.72099167793291
DemoTT_Device101	Automatically created tl	DemoTT	Mon Dec 07 2020 20:5	[object Object]	monkey	0.7003614980385375
DemoTT_Device102	Automatically created tl	DemoTT	Mon Nov 09 2020 01:4	[object Object]	monkey	28.843109052944925
DemoTT_Device103	Automatically created tl	DemoTT	Wed Feb 10 2021 16:0	[object Object]	dog	56.71683284076734

2. Now it's time to add a query to the **QueryDemoThings** service
    a. Edit the service and add a new input called thingName, of type THINGNAME
    b. Add some error handling and a parametrized query
    c. Here is the code:



```
// Put this above the params JSON Object
if(thingName === undefined || thingName === null) {
    // Have to do this or the query will return nothing because thing names cannot
be
    // null or undefined in ThingWorx
```

```
        thingName = "";
    }

    var myQuery = {
      "filters": {
        "fieldName": "name",
        "type": "LIKE",
        "value": "*" + thingName + "*"
      }
    };
```

d. Don't forget to add myQuery to the params, where previously it said undefined:

```
    var params = {
        // This pulls from the properties so as to not be hard-coded in each service
        // This way when more things are eventually added, this number can be increased
        // Leaving this blank will result in only 500 entries being returned
        maxItems: me.MaxItems,
        query: myQuery
    };
    var result =
    ThingTemplates[thingTemplateName].QueryImplementingThingsWithData(params);
```

e. Click "Done" and save the thing

3. Edit the **DemoThingMashup** again

   a. Bind the text target of the text field to the **thingName** input of **QueryDemoThings** (it might not appear without refreshing the mashup)

   b. Bind the button "Clicked" event to the **QueryDemoThings** service

4. Save the mashup and view it

5. See how much slower the **DemoThingMashup** loads when a query is passed in by typing some of a thing name in the new text field and clicking the button; if nothing appears, check the myQuery JSON Object, where you may have put "EQ" instead of "LIKE"

End Section 3: Review

In this section, we learned how to add queries to ThingWorx services. Likewise, we stepped through adding widgets to capture filter information and return search results. Note that in the Advanced Grid widget, this process of sorting and searching is done even more easily (see the appendix for details). Also note that the cut, copy, and paste feature used at the start does not copy cross-mashups, and it can only keep track of the last thing removed, so use it carefully. Duplicate mashups before beginning to make edits to ensure that no working functionality is broken by changes.

Tutorial Section 4: Modify a Thing Template

1. Next, run **DeleteNThings** so we can modify the properties they have and the data shape which displays on the grid (where N = 1000)

a. Since we need to assign the value of the data on thing creation, we have to delete everything first

b. Normally, adding a property does not require all existing things to be deleted, as the property value is updated elsewhere (like from the Edge)

Best Practice Note:
When templates are saved, all implementing things are restarted at once. The restart of all things should be avoided in production, especially if they are remote things receiving data. Saving a thing template in production can cause the entire platform to crash. It is better to import new changes into production outside of business hours, using the built-in import functionality, and not a process of manual changes**.**

2. Modify **DemoTT**

   a. Add two new properties:

      i. LastServiceDate - DATETIME type – persistent

      ii. ID - STRING – persistent

3. Modify the **QueryDemoThings** data shape (**DemoThingDS**)

   a. Add two new fields:

      i. LastServiceDate – DATETIME

      ii. ID – STRING

4. Modify the **DemoAppUtility** thing, **QueryDemoThings** service

   a. Add input parameters for the start date (to), end date (from), and ID

   b. Add more query filters so the final query looks like this (this is not the full service, just the portion with the query):

```
if(thingName === undefined || thingName === null) {
    // Have to do this or the query will return
    // nothing because thing names cannot be null or undefined
    // If the thing cannot be found, we want it to return everything
    thingName = "";
}

if(ID === undefined || ID === null)
    ID = "";

if(to === undefined || to === null)
    to = Date.now();

if(from === undefined || from === null)
    from = "0000-01-01T00:00:00.000-00:00"; // The start of time

var myQuery = {
    "filters": {
        "type": "AND",
          "filters": [
              {
                "fieldName": "name",
                "type": "LIKE",
                "value": "*" + thingName + "*"
              },
              {
                "fieldName": "ID",
                "type": "LIKE",
                "value": "*" + ID + "*"
              },
              {
                "fieldName": "LastServiceDate",
                "type": "BETWEEN",
                "from": from,
                "to": to
              }
         ]
    }

};
```

c. **You have to give defaults to queries, or they will fail to return correct values (do not pass in undefined), and the star symbols are required with the "LIKE" or "CONTAINS" searches**

5. Modify the **CreateNThings** service on the **DemoAppUtility** thing
    a. Add lines to update the datetime with a random value and the ID with a string
    b. Code to add in:

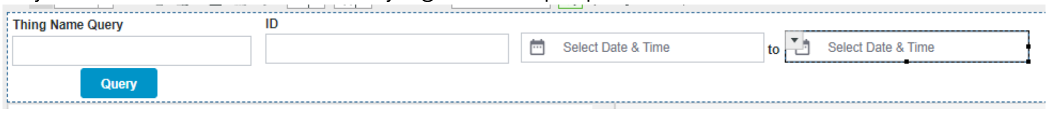
```
// Add this after the other properties are set but before the counter increments
// Convert random number to days by multiplying by 100 to get, then 24 hours per
// day, 60 minutes per hour, 60 seconds per minute, and 1000 milliseconds per second
Things[thingName].LastServiceDate = Date.now() - (Math.random()*100*24*60*60*1000);
Things[thingName].ID = "AutoDT" + i;
```

    c. Full code:

```
var counter = 1;
var thingTemplateName = "DemoTT";

for(var i = 0; i < N; i++) {
    var thingName = "DemoThing";

    try {
        // Each new thing needs a unique name, so either pass it in from a UI,
        // or generate it like this
        thingName = thingName + i;

        // Create the thing //
        var params = {
            name: thingName,
            description: "Automatically created demo thing",
            thingTemplateName: thingTemplateName,
            tags: "Applications:ModelReferenceBestPractice"
        };
        Resources["EntityServices"].CreateThing(params);

        // Need this part because they do not get enabled on their own when created
        // programmatically //
        Things[thingName].EnableThing();
        Things[thingName].RestartThing();


        // Update the thing with initial information //
```

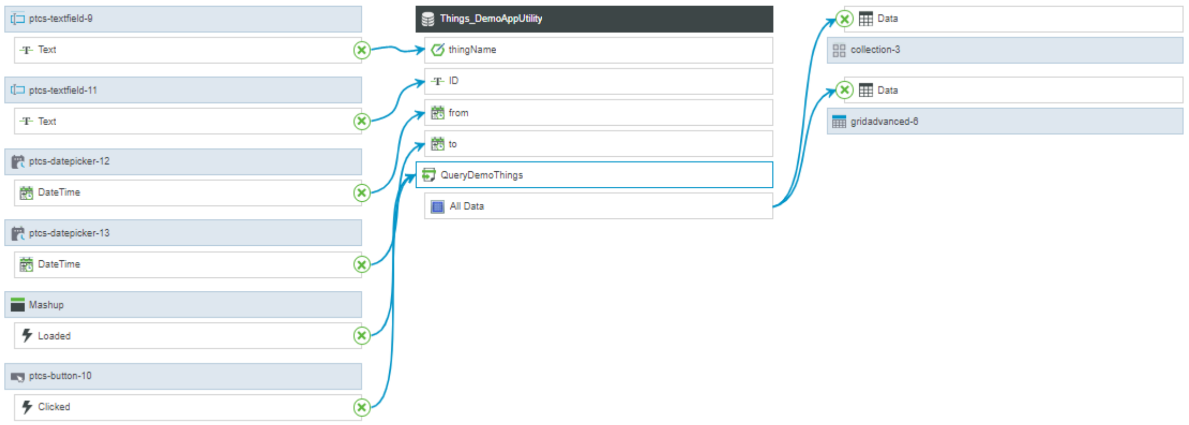
```
                  // Any code can go here, for example, setting initial property values
                  // on the new things //
                  Things[thingName].NumberProp = Math.random()*100;
                  Things[thingName].StringProp = "I've said Hello World " + i + " times.";

                 // Add this after other properties are set but before the counter increments
                 // Convert the random number to days by multiplying by 100 to get a number,
                 // 24 hours per day, 60 minutes per hour, 60 seconds per minute, and 1000
                 // milliseconds per second
                 Things[thingName].LastServiceDate = Date.now()-
                                                         (Math.random()*100*24*60*60*1000);
                 Things[thingName].ID = "AutoDT" + i;

                  counter++;
              } catch(err) {
                 logger.error(me.name + ": CreateNThings: Could not create thing named " +
                     thingName + "; Message: " + err);

                  // As per KCS 198580, you MUST DELETE things created from runtime memory
                  // straight away if they are not created successfully //
                  // Otherwise, you will have ghost entities on the Platform //
                  Resources["EntityServices"].DeleteThing({name: thingName});
              }
          }
          var result = counter;
```

6. Save **DemoAppUtility**
7. Create the things again by calling **CreateNThings** (N = 1000)
8. Modify the **DemoThingMashup**
    a. Add another text field
    b. Add 2 datetime picker widgets
    c. Modify the properties of the datetime picker widgets and deselect the check box for **initializeWithCurrentDateTime** (you may want to change the formats of the dates as well)
    d. Add 3 labels, two which can be configured within the text fields, and 1 which needs to be dragged as a new widget onto the mashup (the "to" label between the datetime pickers)
    e. Lay them out as desired, modifying the text properties and labels, etc.



    f. Bind the ID textbox "text" target to the input property for **QueryDemoThings**

g. Bind the datetime widgets' DateTime target to the to and from fields



h. Save the mashup

9. View **DemoThingMashup**

   a. Play around with the filters and datetime boxes to see how they work

   b. Things should run slower, but still not be too bad

End Section 4: Review

In this section, we learn how to add more complex queries to data sources, and how to modify thing templates after they have already been implemented. Note the importance of never saving a thing template in a production environment. It has the potential to cause outages and data loss on the Foundation server (see appendix). We also learned how to generate dummy dates using random numbers in ThingWorx, which is simple since the random functionality is built right into JavaScript.

Tutorial Section 5: Info Table Properties and Grids

Best Practice Note:
Not including info tables in data shapes will ensure mashups never receive them. Rendering info tables in grids can be very slow, depending on the number of lines. For similar reasons, never mark an info table as logged. Info tables are very memory intensive and should be considered "expensive".

1. Next, delete all things again using **DeleteNThings** on the **DemoAppUtility** thing (N = 1000)

2. Create another data shape called **InfotablePropDS** with two fields, "Field1" and "Field2", both NUMBER type properties (this is a new DS separate from existing ones)

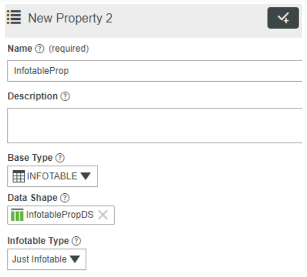
3. Modify **DemoTT**

   a. This time do **NOT** modify the **DemoThingDS**

   b. Add an info table property with the data shape **InfotablePropDS** called **InfotableProp**

   c. Save the **DemoTT** thing template

4. Modify **DemoAppUtility**, **CreateNThings** service


New Property 2

Name (required)
InfotableProp

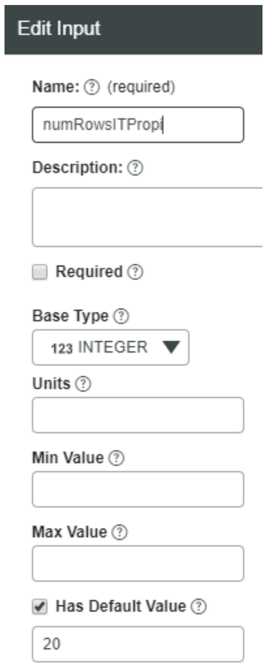
Description

Base Type
INFOTABLE

Data Shape
InfotablePropDS

Infotable Type
Just Infotable

a. Add an input parameter with type INTEGER: **numRowsITProp** (give it a default value of 20)


Edit Input

Name: (required) numRowsITProp

Description:

Required: ☐

Base Type: INTEGER

Units:

Min Value:

Max Value:

Has Default Value: ✓

20

b. Add code to create and populate the info table with random data, and place it under the other property updates:

```
// Create an infotable with the InfotablePropDS //
var it =
Resources["InfoTableFunctions"].CreateInfoTableFromDataShape({
   infoTableName : "InfoTable",
   dataShapeName : "InfotablePropDS"
});

// Populate fields with random numbers and add several rows //
for(var j = 0; j < numRowsITProp; j++) {
      var newEntry = new Object();
      newEntry.Field2 = Math.random()*100;
      newEntry.Field1 = Math.random()*100;
      it.AddRow(newEntry);
}

Things[thingName].InfotableProp = it;
```

c. The full service code should now look like this:

```
var counter = 1;
var thingTemplateName = "DemoTT";

for(var i = 0; i < N; i++) {
    var thingName = "DemoThing";
    try {
        // Each new thing needs a unique
        // name, so either pass it in from a
        // UI, or generate it like this
        thingName = thingName + i;

        // Create the thing //
        var params = {
            name: thingName,
            description: "Automatically created demo thing",
            thingTemplateName: thingTemplateName,
            tags: "Applications:ModelReferenceBestPractice"
        };
        Resources["EntityServices"].CreateThing(params);

        // Need this part because they do not get enabled on their own when created
        // programmatically //
        Things[thingName].EnableThing();
        Things[thingName].RestartThing();

        // Update the thing with initial information //
        Things[thingName].NumberProp = Math.random()*100;
        Things[thingName].StringProp = "I've said Hello World " + i + " times.";

       // Convert random number to days by multiplying by 100 to get a number, then
       // 24 hours per day, 60 minutes per hour, 60 seconds per minute, and 1000
       // milliseconds per second
        Things[thingName].LastServiceDate = Date.now()-(Math.random()
               *100*24*60*60*1000);
        Things[thingName].ID = "AutoDT" + i;

       // Create an infotable with the InfotablePropDS //
```

```
            var it = Resources["InfoTableFunctions"].CreateInfoTableFromDataShape({
               infoTableName : "InfoTable",
               dataShapeName : "InfotablePropDS"
            });

            // Populate the fields with random numbers and add several rows //
            for(var j = 0; j < numRowsITProp; j++) {
                var newEntry = new Object();
                newEntry.Field2 = Math.random()*100;
                newEntry.Field1 = Math.random()*100;

                it.AddRow(newEntry);
            }

            Things[thingName].InfotableProp = it;
            counter++;
        } catch(err) {
             logger.error(me.name + ": CreateNThings: Could not create thing named " +
                     thingName + "; Message: " + err);

             // As per KCS 198580, you MUST DELETE things created from runtime memory
             // straight away if they are not created successfully //
             // Otherwise, you will have ghost entities on the Platform //
             Resources["EntityServices"].DeleteThing({name: thingName});
         }
    }
    var result = counter;
```

5. Run **CreateNThings** to create things with info table properties
6. Open the **DemoThingMashup** mashup and see how long it takes to query now
    a. Open the “Platform Subsystem” and see how the memory in use changes between scenarios
        i. Go to “Monitoring > Subsystems” in the navigation bar
        ii. When the things are deleted, the memory use climbs as the GC cleans out the JVM, and then drops accordingly
        iii. When things are created, the amount of memory used climbs depending on the number of things and the number (and size) of info tables per thing, since info tables eat up a lot of the JVM’s memory and are very costly (See the appendix for details)
    b. Call **DeleteNThings** for N things
    c. Set **numRowsITProp** to 100 and run **CreateNThings** to see how this affects performance
    d. Modify the **InnerDemoMashup** to display the **InfotableProp** in a grid, and note that performance is even slower (this modification is not included in the mashup provided here, as this is not a good design choice and is only for demonstrating how runtime is effected)
    e. Call **DeleteNThings** again, and then call **CreateNThings**, but this time add 2000 things, still with info table properties as configured in step c (note that more than this will not be queried unless **DemoAppUtility.MaxItems** is increased)

f. Refresh the mashup and see how the mashup loading affects the memory (in bytes) on the Platform Subsystem (memory usage should go up a bit)

g. Now, open the same mashup in two different browsers, refresh them at the same time, and then observe the effect on memory (memory usage should increase substantially)

7. Note that this trend will continue

a. The more properties you have, particularly memory intensive ones like info tables (especially those with more than a hundred rows), the worse this query will perform

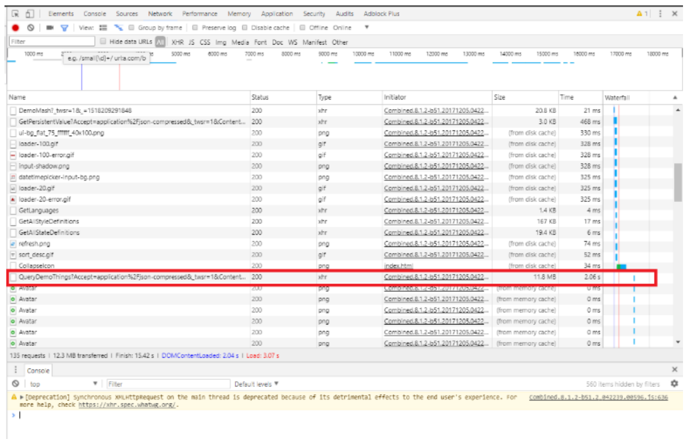
b. This is especially so once data ingestion is added into the equation.

i. Remote properties being updated by the Edge every 30 seconds or so (sometimes less) will result in more limited access to these properties

ii. More things, more properties, more bandwidth, worse performance

iii. **Browser Tools** (hit F12 in most browsers) helps identify mashup performance issues

iv. Especially useful is the **Network** tab, which shows which services were called, and when they completed, as well as their status codes, payload information, request headers, and other details about each request and response



Name	Status	Type	Initiator	Size	Time	Waterfall
DemoMash?_twsr=1&_=1518209291848	200	xhr	Combined.8.1.2-b51.20171205.0422...	20.8 KB	21 ms	
GetPersistentValue?Accept=application%2Fjson-compressed&_twsr=1&Content...	200	xhr	Combined.8.1.2-b51.20171205.0422...	3.0 KB	468 ms	
ui-bg_flat_75_ffffff_40x100.png	200	png	Combined.8.1.2-b51.20171205.0422...	(from disk cache)	330 ms	
loader-100.gif	200	gif	Combined.8.1.2-b51.20171205.0422...	(from disk cache)	328 ms	
loader-100-error.gif	200	gif	Combined.8.1.2-b51.20171205.0422...	(from disk cache)	328 ms	
input-shadow.png	200	png	Combined.8.1.2-b51.20171205.0422...	(from disk cache)	328 ms	
datetimepicker-input-bg.png	200	png	Combined.8.1.2-b51.20171205.0422...	(from disk cache)	325 ms	
loader-20.gif	200	gif	Combined.8.1.2-b51.20171205.0422...	(from disk cache)	325 ms	
loader-20-error.gif	200	gif	Combined.8.1.2-b51.20171205.0422...	(from disk cache)	325 ms	
GetLanguages	200	xhr	Combined.8.1.2-b51.20171205.0422...	1.4 KB	4 ms	
GetAllStyleDefinitions	200	xhr	Combined.8.1.2-b51.20171205.0422...	167 KB	17 ms	
GetAllStateDefinitions	200	xhr	Combined.8.1.2-b51.20171205.0422...	19.4 KB	6 ms	
refresh.png	200	png	Combined.8.1.2-b51.20171205.0422...	(from disk cache)	74 ms	
sort_desc.gif	200	gif	Combined.8.1.2-b51.20171205.0422...	(from disk cache)	52 ms	
CollapseIcon	200	png	index.html	(from disk cache)	34 ms	
QueryDemoThings?Accept=application%2Fjson-compressed&_twsr=1&Content...	200	xhr	Combined.8.1.2-b51.20171205.0422...	11.8 MB	2.06 s	
Avatar	200	png	Combined.8.1.2-b51.20171205.0422...	(from memory cache)	0 ms	
Avatar	200	png	Combined.8.1.2-b51.20171205.0422...	(from memory cache)	0 ms	
Avatar	200	png	Combined.8.1.2-b51.20171205.0422...	(from memory cache)	0 ms	
Avatar	200	png	Combined.8.1.2-b51.20171205.0422...	(from memory cache)	0 ms	
Avatar	200	png	Combined.8.1.2-b51.20171205.0422...	(from memory cache)	0 ms	
Avatar	200	png	Combined.8.1.2-b51.20171205.0422...	(from memory cache)	0 ms	

135 requests | 12.3 MB transferred | Finish: 15.42 s | DOMContentLoaded: 2.04 s | Load: 3.07 s

```
▶ [Deprecation] Synchronous XMLHttpRequest on the main thread is deprecated because of its detrimental effects to the end user's experience. For    combined.8.1.2-b51.2.042239.00596.js:636
more help, check https://xhr.spec.whatwg.org/.
>
```

v. If the service has not returned after several seconds or more, then it may need to be optimized. Also consider using data tables or streams in place of info tables, storing data somewhere other than on the things themselves if it needs to be modified by users or queried frequently. If the query returns, and yet the data takes a while to render on the mashup, then this is likely not performance related, but widget rendering related (so consider opening a TS case).

End Section 5: Review

In this section, we learned how to generate info tables with random information, and we observed the effect of info table properties on overall system performance. Note that the need for limiting the size of info tables is directly related to the amount of memory available to the JVM. Purge mechanisms need to be in place to keep data low, whether it is purged when new data comes in, or periodically in a timer subscription. Best practice is to limit info table properties to under 100 rows, using them to store static information about the server (like the names of different types of assets) or information only needed for short periods of time, during service execution.

We also learned how to review status information about memory usage on the Platform in the subsystem monitor, as well as how to use Browser Tools to determine if services are completing in a timely fashion. The first step to troubleshooting any sort of performance issue should be to open Browser Tools and look for service completion times. To be certain, logging statements can also be added to the start and end of every service so that the exact time stamps appear in the logs.

Example Completion - Summary:

This concludes the first example in this document. You should now know how to:

- Create and delete thousands of things easily and programmatically without creating ghost entities in the process
- Generate and assign dummy data to thing properties
- Create mashups to display this data, using both collections and grids
- Create and utilize mashup parameters (which are more performant than session variables, as discussed in the appendix)
- Create queries which take in various types of search criteria
- Efficiently query against many things at once on an overview of things mashup
- Create the UI necessary for enabling filtering in a grid

Example 2: Isolating Data Ingestion from User Influence

- **Teaches how to:**
  - **Safely access a data table**
  - **Fuel mashups with pre-processed data sources to improve performance**
  - **Simulate data updates using random numbers**
  - **Write queries for data tables**
  - **Index fields for data tables**
- **Covers:**
  - **Developing a more complex and stable data model**
  - **Separating the data ingestion process from the data rendering on mashups**
  - **Using timers to fuel updates on the Foundation server**
  - **Using expressions to build more complicated mashups**

Context

All approaches to querying data on mashups boil down to the same ideology: don't pull or modify the data coming in from the Edge as it comes in, i.e. separate the use of the data from the ingestion of the data, wherever possible. Sometimes, if there is enough data, querying on mashups can result in poor performance. There are many solutions to this problem, but the one discussed here involves populating a data table (which can have its own downsides, so consider the overview in the introduction carefully).

Say that the end requirement was to have a table showing overview information about each asset. The data doesn't need to be 100% real-time, refreshing to within a minute or so, for about 2000 things. End users (mechanics and admins) would then need to be able to query the connected assets, and sort them by which are theirs on the mashup. Here, we need to design a mechanism which periodically updates the data table with fresh information from new and existing entities. We only want to give the mashup exactly what it needs to load the grid in an effort to limit any unnecessary use of memory. The separation of the ingestion from the data usage here will ensure that nothing can damage the stability of the data coming in, which should be considered a very high priority for most customers, because data loss often corresponds with revenue loss.

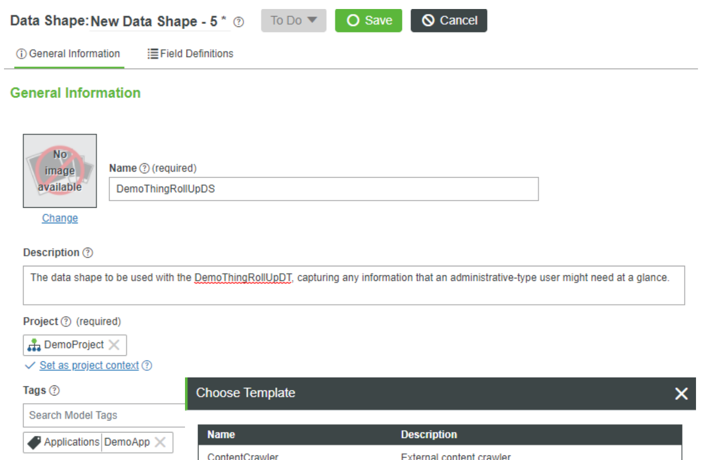
Note that this approach is not real-time. Typically, there would be separate, much simpler alarm protocols tied to remote property change events for real-time notifications (covered in Example 3). If a real-time display of many, many things at once is needed, then one option is to consider using an optimized ingestion application, like Kepware or InfluxDB. Other options include limiting how much data a mashup can retrieve at a time while still using the ingested data source, or only allowing users to view real-time data for one thing at a time. Also consider limiting the amount of information coming in from the Edge, redesigning based around the question "how much data do we really need?"

In line with this same question, consider using separate streams to aggregate the data (i.e. to reduce it by some common algorithm, often called "smoothing"). These can be updated periodically such that more historical data can be kept for longer, and with fewer existing data points to slow down query and mashup performance. This is a common mechanism found throughout database architecture.

Finally, rely upon listing thing names (using the much faster GetImplementingThings without data call) instead of real-time data on overview mashups. Selecting from this list can then load another mashup, where GetProperties is used to show real-time updates for one thing at a time . Rarely does a human user need to know real-time info for all of the assets at once in an IoT application.

Tutorial Section 1: Create Data Shapes and Data Tables

1. Start by creating a data shape, with a primary key (required for data tables) called **DemoThingRollUpDS**


Data Shape: New Data Shape - 5 *

General Information | Field Definitions

General Information

Name (required): DemoThingRollUpDS

Description: The data shape to be used with the DemoThingRollUpDT, capturing any information that an administrative-type user might need at a glance.

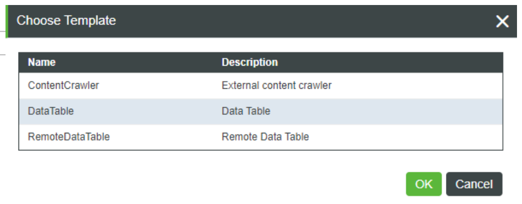
Project (required): DemoProject

Set as project context

Tags: Search Model Tags

Applications | DemoApp

2. Add whatever fields you wish to appear in the grid on the mashup
   a. In this example, we will use only the custom fields we created previously (not the info tables)
   b. Be sure to check "Is Primary Key" for the ID field
3. Create a data table called **DemoThingRollUpDT** which uses this data shape


Choose Template

Name	Description
ContentCrawler	External content crawler
DataTable	Data Table
RemoteDataTable	Remote Data Table

   a. Choose the default type of table
   b. Fill out the name and tags
4. Add a service to this data table for updating entries, with any custom logic required
   a. Call it **CustomAddOrUpdate**, with inputs for each of the properties to display on the mashup (ID, DateIn, StringIn, NumberIn, and NameIn as seen below), and the result type set to BOOLEAN
   b. Here is the code:

```
var success = -1;

// Don't want invalid dates //
if(DateIn === null || DateIn === undefined)
    DateIn = Date.now();

// Need a valid ID, or else this entry is new //
if(ID === null || ID === undefined || ID === "")
    ID = "ManualDT" + dateFormat(DateIn, "mmddyy-hhmmss");

// Don't want negative numbers //
if(NumberIn === null || NumberIn === undefined || NumberIn < 0)
    logger.warn(me.name + ": Cannot update/add entry with ID \"" + ID
            + "\"; Message: NumberIn cannot be negative.");
else {
    var values = me.CreateValues();

    values.LastServiceDate = DateIn; //DATETIME
    values.StringProp = StringIn; //STRING
    values.ID = ID; //STRING [Primary Key]
    values.NumberProp = NumberIn; //NUMBER

    var params = {
        sourceType: "STRING",
        values: values,
        // This field is also indexed, and we want to search by name later
```

```
            source: "ServiceAdded: " + NameIn
        };
        try {
             success = me.AddOrUpdateDataTableEntry(params);
        } catch(err) {
             logger.error(me.name + ": Could not add or update entry with ID " + ID
                 + "; Message: " + err);
        }
    }
    var result = false;

    if(success >= 0) {
        // Avoid making statements like this "info" level or lower, as blowing up the
        // logs with statements can hurt performance
        logger.debug(me.name + ": Successfully added new entry with ID: " + ID);
        result = true;
    } else {
        logger.warn(me.name + ": Failed to add new entry with ID: " + ID);
    }
```

   c. Change this to suit the use case of each, what values are and are not permitted (for instance, a check to see if the date is in the past or the future could be added, etc.)

5. Create another service called **UpdateAllDataTableEntries** on the same **DemoThingRollUpDT** thing

   a. This service should call the other service a number of times, based on what input is given

   b. In this case, we are assuming the data is coming from Edge devices, and we just want to display the most recent values on each device and return the total number updated

   c. We could just as easily want to calculate or process the data in some way, and this would be the place to do that, with more fields added to store any such new information

   d. Here is a script which pulls up each thing in turn and merely copies its properties into the table using the service we just wrote, then returning how many rows were updated:

```
 // This returns faster than QueryImplementingThingsWithData, and allows us to still
 // retrieve the data off every thing with no arbitrary, hard-coded limits. Doing it
 // this way instead of looping through the table ensures new things get added
 // Consider adding thingTemplateName as a parameter as seen in the first example
 var allThings = ThingTemplates["DemoTT"].QueryImplementingThings({
                              maxItems: Things["DemoAppUtility"].MaxItems});

var counter = 0;
for(var i = 0; i < allThings.rows.length; i++) {
    var thing = Things[allThings.getRow(i).name];

    var success = me.CustomAddOrUpdate({DateIn: thing.LastServiceDate, NumberIn:
     thing.NumberProp, ID: thing.ID, StringIn: thing.StringProp, NameIn: thing.name});

     if(success)
        counter++;
  }

  var result = counter; // The number of things added or updated successfully
```

```
// Note that this service does not account for things being deleted
// The best thing for runtime is to have whatever mechanism deletes the thing itself
// to remove it from the DT. Alternatives include implementing a purge mechanism, or
// appending this service to include such logic (but the best place for it is from a
// mashup dedicated solely to deleting assets from TWX)
```

End Section 1: Review

In this section, we learned how to create data shapes and data tables. Note that data tables are different than info tables in that their data shapes need to have a primary key specified. This means that in addition to the source column being indexed, a second field can be indexed to improve search performance. We also learned how to create services to update individual rows, or the entire data table at once. There is no purge mechanism built here, because ideally, whatever service deletes the thing from the Foundation server would also remove it from the data table. Since mashups allowing for thing creation and deletion are not created as a part of this tutorial, details on how to do this particular type of purge have been left out.

Note that if the application is very complex, requiring multiple data tables, and updates like this on each, then it may be better to use relational databases instead of built-in data sources, or to generally rethink the design of the overall application. This solution only works because the application is very simple.

Best Practice Note:
Subscriptions which run **on a thing** will result in higher memory usage in the **Event Processing Subsystem**, which is typically also responsible for data ingestion. Subscriptions **on timers themselves** hit the **Platform Subsystem**, pulling from a much larger pool of memory in a subsystem which usually has much less to do.

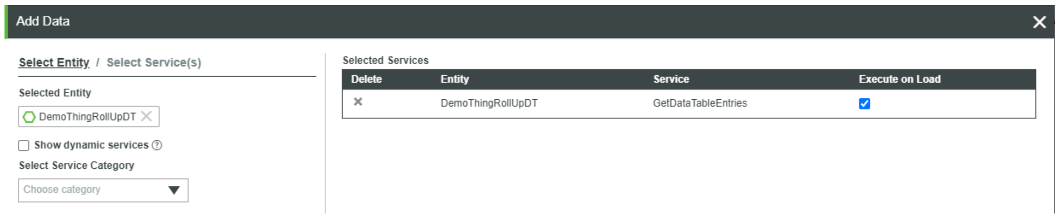
Tutorial Section 2: Create Timer Subscriptions

1. Now we need to create a timer thing (**DemoRollUpRefreshTimer**)
    a. Create a new thing and use the Timer thing template
    b. Here, we will have the timer subscribe to itself, so go to the Subscriptions tab and click "Add" (call it "**DataRefresh**")
    c. Under the Event tab, select "Timer" for the event type
    d. Here is the code:

```
logger.trace("Entering UpdateAllDataTableEntries. Updating DemoThingRollUpDT...");
var updatedCount = Things["DemoThingRollUpDT"].UpdateAllDataTableEntries();
logger.debug("Updated " + updatedCount + " things in DemoThingRollUpDT.");
```

    e. This is a basic amount of logging helpful for debugging purposes
        i. Note that there were also errors and warnings throughout the contained services too
        ii. More on "smart logging" to come (in Example 3)
    f. Be sure to check the box to enable the subscription on the "Subscription Info" tab

2. Swap back to the "General Information" tab and scroll down to updated the "Run As User" to Administrator (for now, since this is just a demo app) and "Update Rate" to 30 seconds
    a. Change the 60000 milliseconds (the default) to 30000 milliseconds
    b. Note that if the data has to be processed or reformatted, say if there are complex calculations or queries against historical data required here, then this value may need to be increased to account for that
    c. If there are many, many devices, consider using more than one data table like this, one per each thing template, and one thing template per region or per device type, whatever makes the most sense, to keep the data stores small enough for updates and queries to complete most quickly
    d. It is always better to separate the ingestion (how the data is brought into the Foundation server) from the services which populate the mashups, meaning that remote service calls here should be avoided at all costs; remote service calls can take ages, even with timeouts configured
    e. Write the query logic and use print statements at the start and end to see how long it takes to run on average (be sure to do this in an environment with a load comparable to Production). If the service takes longer than 30 seconds to complete at any time, be sure to reconfigure this timer so updates do not overlap (or deadlocks can occur)
3. Now, open **DemoThingMashup** from the previous to add the new service
    a. Add the service **GetDataTableEntries** from the **DemoThingRollUpDT**



Selected Entity: DemoThingRollUpDT

Selected Services

Delete	Entity	Service	Execute on Load
×	DemoThingRollUpDT	GetDataTableEntries	✓

    b. Bind it to the grid on the right-hand side of the screen (deleting the other binding)
    c. Click View Mashup and note how much quicker the grid loads over the collection
    d. Note that a little extra work is required to get the query filters working for this, namely a wrapper service like **QueryDemoThings** which queries against the data table instead of all things
4. None of the data changes at present, so we can't see if refresh on the grid side. Next, we will add a timer to generate random values for these fields so that the updates can be seen here

End Section 2: Review

In this section, we learned how to create timers and subscribe to their timer events. Subscriptions to timer events which occur on the timer itself hit a different memory pool in their service execution than those located on other entities. This is important because updating sources of data can sometimes take a few minutes to complete. Having the Platform Subsystem handle these changes ensures that the data ingestion process, ordinarily centered around property updates and event triggers, can continue uninterrupted by data reorganization or aggregation processes (as the Stream or Value Stream Subsystem handle the ingestion, and the Event Subsystem, the alerts). We also saw that retrieving data from a data table is faster than querying all implementing things, and this effect is more pronounced the more things and the more properties per thing which exist on the Foundation server. Even though we didn't implement a query against the data table in this example, it is recommended to do so, creating a wrapper service that uses the same query creation logic as **QueryDemoThings**. Then the same mashup fields can be used to query against the data table too, showing directly that the data table is faster.

Tutorial Section 3: Randomize Data Updates

Best Practice Note:
If updates to local properties are required, update them when the Edge properties update as data change subscriptions, only setting the data change to occur when the value actually changes on the Edge property ("value"). If all devices need to be updated at once, use a scheduler to do the change outside of business hours.

1. To add randomized data updates, we will create a thing template level timer
    a. Since we are only updating local properties, or those located on the thing with the subscription, this is relatively safe for a demonstration app.
    b. It is always safer to push changes from the Edge than to pull them from the Foundation (see the appendix).
2. Add a service to **DemoTT** called **UpdateLocalPropertiesRandom**:

```
logger.debug("Updating local properties for " + me.name);

try {
    // Random strings for the StringProp
    var stringArr = ["dog", "cat", "monkey", "elephant", "pig"];

    // Choose a random index from 0 to the size of the array above for the StringProp
    var index = Math.floor((Math.random() * stringArr.length));
    me.StringProp = stringArr[index];

    // Choose any random number for the NumberProp
    me.NumberProp = Math.random()*100;

    // Choose any random day in the past 60 days for LastServiceData (convert the random
    // number to days too)
    me.LastServiceDate = Date.now() - Math.floor(Math.random()*60)*24*60*60*3600;
    // The ID field does not vary, and is assigned only once when the thing is created
    // So the updates are done!
} catch(err) {
       logger.error("Could not update properties for " + me.name + "; Message: " + err);
}

logger.debug("Successfully updated local properties for " + me.name);
```

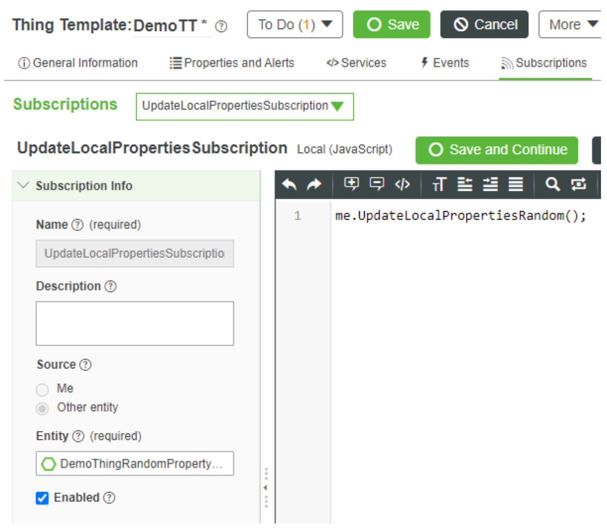
3. Add a new thing using the timer template again (called **DemoThingRandomPropertyUpdateTimer** here), with the same 30000 milliseconds and "Run As User" configured
4. Back on the **DemoTT** thing template, add a subscription
    a. On the "Subscription Info" tab, check the "Other entity" option

b. Supply the name used for the timer in the previous step and be sure to check the enabled check box

c. Once you supply the name of the thing, switch to the Inputs tab and select "Timer" in the drop-down box

d. The only code in this subscription is a line to call the service we just wrote: `me.UpdateLocalPropertiesRandom();`

e. Now, every 30 seconds, each device will update its properties



Name (required): UpdateLocalPropertiesSubscriptio

Description:

Source: Me / [✓ Other entity]

Entity (required): DemoThingRandomProperty...

☑ Enabled

```
me.UpdateLocalPropertiesRandom();
```

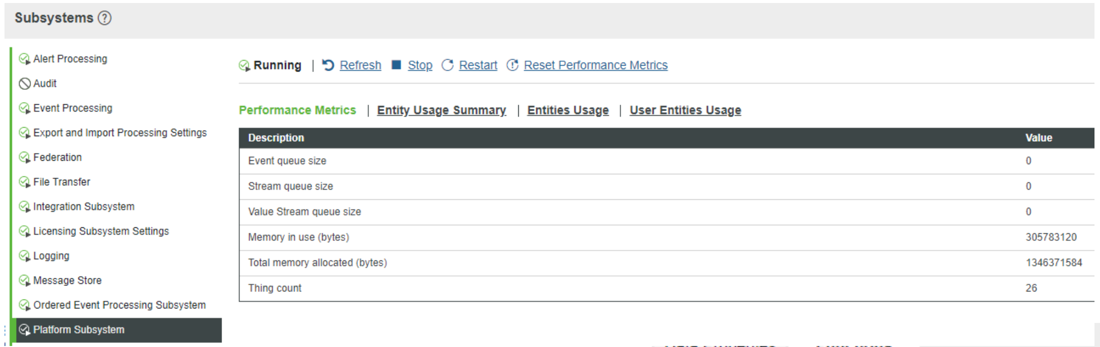
i. Note that this technique should **rarely** be employed in a real application

ii. These are quick updates to local properties in a contained demo system, but consider alternative approaches for a production system, like pushing updates from the Edge as needed, only when the values actually change

f. Review how this additional service affects performance

i. Look back at the Platform Subsystem once this subscription is enabled to see how it affects the system (**Monitoring > Subsystems > Platform Subsystem**)

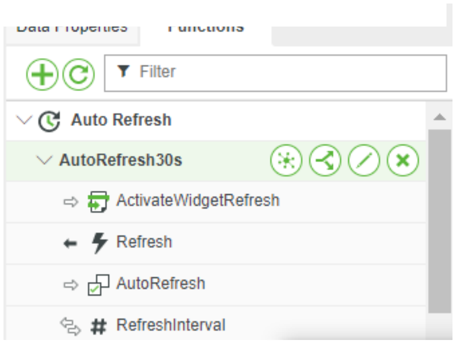
ii. The memory usage spikes very high when the updates are performed, and drops once the garbage collection has completed


Subsystems
Alert Processing
Audit
Event Processing
Export and Import Processing Settings
Federation
File Transfer
Integration Subsystem
Licensing Subsystem Settings
Logging
Message Store
Ordered Event Processing Subsystem
Platform Subsystem
Running | Refresh | Stop | Restart | Reset Performance Metrics
Performance Metrics | Entity Usage Summary | Entities Usage | User Entities Usage

Description	Value
Event queue size	0
Stream queue size	0
Value Stream queue size	0
Memory in use (bytes)	305783120
Total memory allocated (bytes)	1346371584
Thing count	26

5. Back on the **DemoThingMashup**, add an auto-refresh function in the panel under the data panel on the bottom right side of the screen, binding its Refresh event to the data source for the grid

6. View the mashup again, and watch for it to auto-update the data after 30 seconds

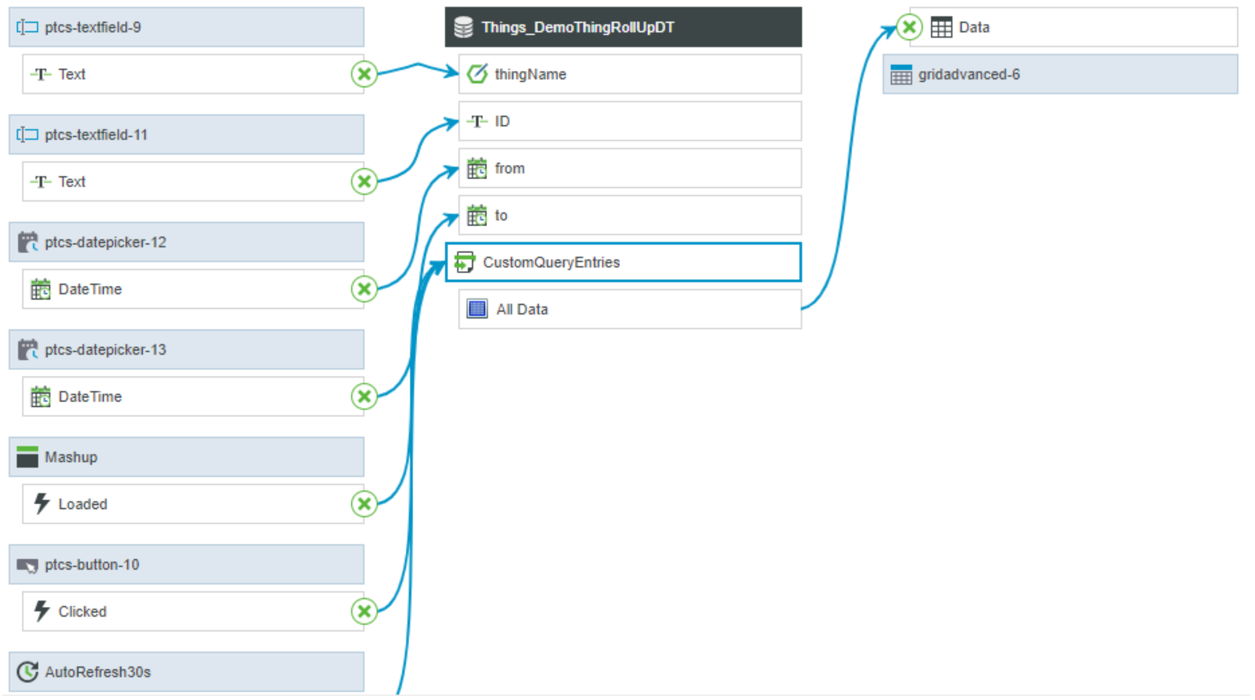


a. Note that the data will automatically update in the collection because of the GetProperties service and its option to auto update, which we enabled previously

b. Notice that one refresh is real-time, and the other on a 30 second interval

   c. The real-time updates are more memory intensive, and may see performance issues if there are many end users, many things, and many properties per thing

7. Remember that the query filters will not work unless you create a wrapper service, called **CustomQueryEntries** on the data table provided with the entities here. The query is the same, and so are the bindings, save for the auto refresh which the collection doesn't need:



   a. In order to allow for partial matches of thing names in searches, we have to rely on a "like" query applied after the data table entries are retrieved, which can be costly and memory intensive. If we wanted, however, we could use the source name to pinpoint a specific entry by name without having to perform a query at all

   b. Use the source field in this example by searching for "ServiceAdded: <thingName>" if a specific thing is needed; this is a great option if manual user updates to one row at a time are possible, especially considering that data tables lock and unlock one row at a time

Best Practice Note:
Any filter sent into a query inside of the query parameter itself will not be applied until after the query returns its results. This results in higher memory consumption per user on the mashup, resulting in potential service crashes. Send in a valid source, to date, and from date, or a max items, to limit results.

End Section 3: Review

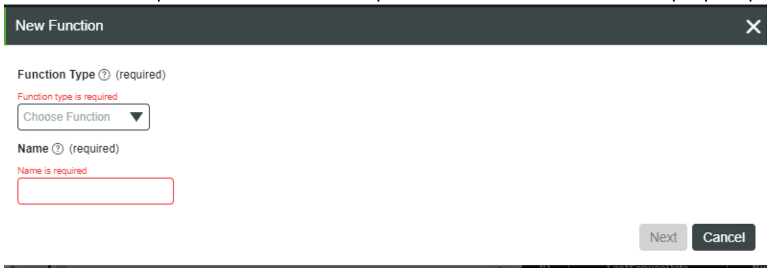
In this section, we learned how to simulate random property updates for many things. We also learned how to implement timer subscriptions on the thing template level, and why it is normally better to avoid such implementations, instead pushing property changes from the Edge (and not pulling them from the Foundation server). We also learned how to check the performance of periodic updates by reviewing the way the memory is consumed in the subsystem monitor. Finally, we covered how to use queries with data tables, and the importance of prioritizing the properties that get sent into a query service itself (like the source field) over the query string (since it's applied after the data is returned).

Note that the Advanced Grid widget has a sort capacity of its own which can be used to limit what appears, but not what is returned into the table. If for some reason both of these are desired side-by-side, then simply pass in the query parameter from the Advanced Grid (called QueryFilter) to the service, and if it has a value, use that input instead of the inputs from the other widgets.

Sometimes, the query needs to be checked in advance of being sent to the data source, because some specific formatting is required for the input. Sometimes, an error message is desired when the formatting isn't given correctly, even before the query button is clicked. This error could be made to appear on the screen or show up in a pop-up box, all in an effort to help the user fill out the form. Building something like this requires using mashup functions to check the data in the text box widgets before it is sent into the query service, as the next example shows.

Tutorial Section 4: Add Pop-Up Errors to Mashup

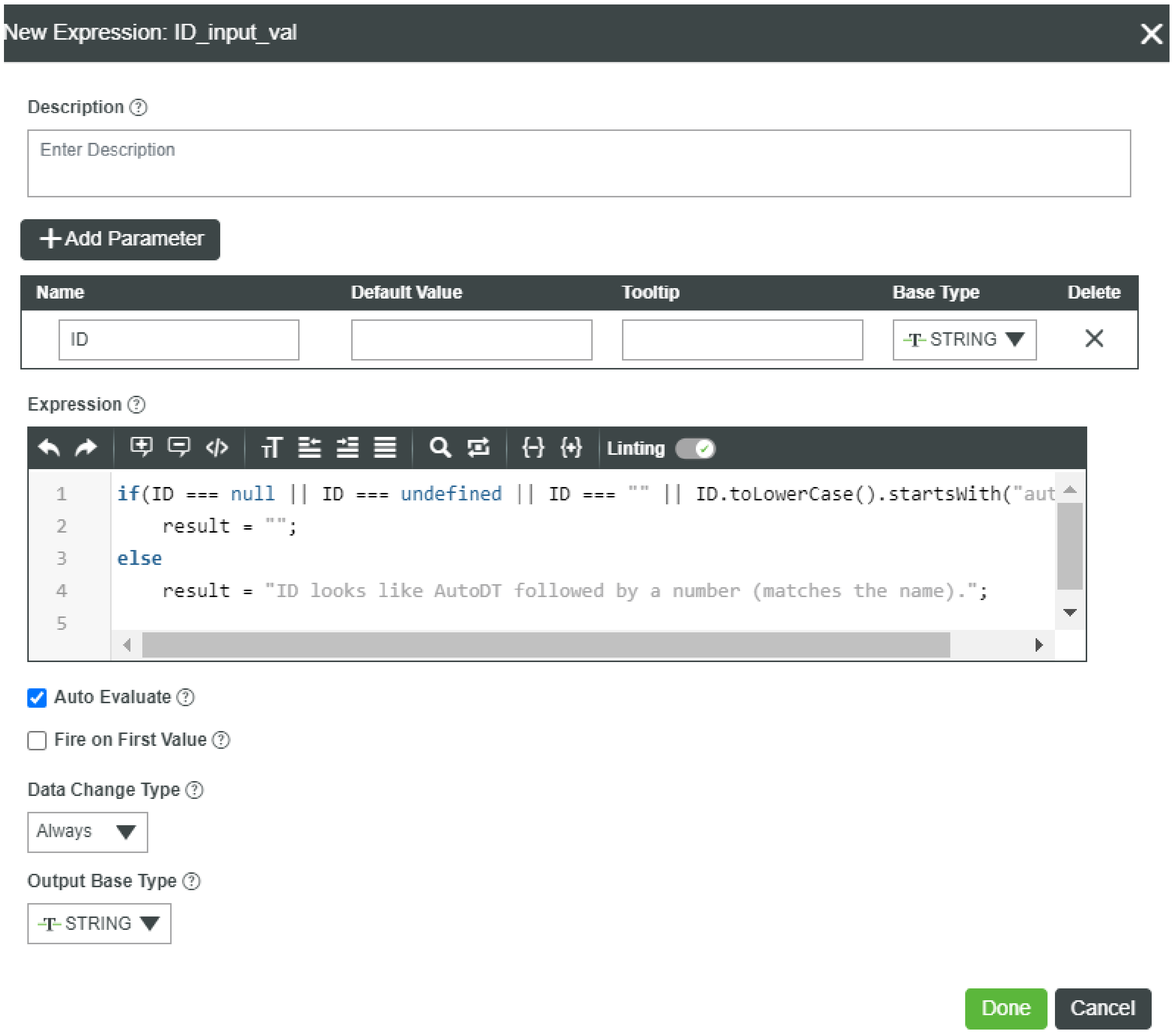
1. Open the **DemoThingMashup** and in the data panel, click on "Functions"
    a. Click on the plus icon at the top of the little box, and a pop-up appears



    b. Select "expression" for "Function Type", and name it **ID_input_val**
    c. Add a parameter for the ID text field input
    d. Since the IDs are all assigned with the same format, we can verify the given ID has that correct format, so here put an expression like:

```
if(ID === null || ID === undefined || ID === "" ||
                                        ID.toLowerCase().startsWith("autodt"))
    result = "";
else
    result = "ID looks like AutoDT followed by a number (matches the name).";
```

    e. Check "Auto Evaluate", change the "Data Change Type" to "Always", change the "Output Base Type" to "String"



Name	Default Value	Tooltip	Base Type	Delete
ID			STRING	✕

Expression

```
if(ID === null || ID === undefined || ID === "" || ID.toLowerCase().startsWith("aut
    result = "";
else
    result = "ID looks like AutoDT followed by a number (matches the name).";
```

Auto Evaluate [✓]

Fire on First Value [ ]

Data Change Type: Always

Output Base Type: STRING

f. Click “Done”

g. Bind the “Text” target of the ID textbox to the input parameter of the expression

2. Add a label widget to the mashup, below the date time selectors

   a. Make sure that the label is long enough to fit the whole message once it appears

   b. Bind the output of the expression above to label widget’s Text target

3. View the mashup and type in the ID text field to see the message appear

   a. In older versions, it may not appear the minute someone begins typing, as the box wasn’t considered updated until it lost focus.

   b. Things like this can always be modified with custom events using ThingWorx widget extensibility (with details found in the ThingWorx Extension Development Guide) The message should only go away when a person types “AutoDT” (not case sensitive) into the box or erases its value completely

   c. Notice how clicking on the Query button will cause the message to appear to help a person know why the data they entered is not valid as well

End Section 4: Review

In this example, we learned how to use an expression function to make an error message or formatting tip pop-up on the screen. This expression function, as well as the validator function, can be used to validate input from text fields, calculate whether or not to display certain labels (such as error messages or confirmations), or determine if other widgets should appear at all (for instance to determine that a drop-down list should only display if a certain option is checked). These functions enable the creation of interactive and intelligent mashups, centered around the end user experience.

Example Completion – Summary:

This concludes the second example in this document. You should now know how to:

- Create timers
- Subscribe to timers on either the thing template level or the timer itself
- Determine when it is a bad idea to subscribe to a timer on a thing template
- Create data tables and data shapes for data tables (different than for info tables)
- Index data tables using the primary key and source fields
- Build queries for data tables
- Implement random data generation mechanisms
- Check on system performance after implementing a new data table update mechanism
- Use the expression widget on a mashup to alter or validate what gets displayed

Example 3: Smart Logging and Alerting

- **Teaches how to:**
  - **Create events, subscriptions, and alarms (things which notify end users via email)**
  - **Utilize the Mail Extension**
  - **Add events to streams**
  - **Build a basic rules engine for alarms**
- **Covers:**
  - **Creating a data shape for an event**
  - **Subscribing to a data change event vs. a manually created event**
  - **Configuring the Mail Extension to use a Gmail account**

Context:

If something goes wrong in the application, and all that happens is a debug statement gets printed to the logs, then the end users still don't really know what is going on. These business-side users may or may not have access to the logs, and they may or may not have the ability to parse through them. In this case, system administrators would be needed. They could turn up the logging if need be to ensure debug statements are captured in the logs, then review the logs to see what is going on. This is an arduous and intensive process, and one which will almost certainly also involve technical support.

For some types of failures, it may be an option to reduce the level of the logging so that instead of printing as a debug statement, the issue is printed as an error or a warning instead. However, if something fails often, it can spam the logs and make it impossible to find anything else (potentially hiding the more important issues from view), also leading to a support nightmare. To avoid situations like this, it is prudent to use smarter logging methods which don't involve getting technical assistance, or even system administrators, at all.

For some types of failures, this will involve pop-up error messages or alerts, or basic things which the user can react to on the UI. But what about errors that happen in the background, like issues updating entries in a data table every few minutes? It wouldn't make sense to have pop-up errors for these, so how can we alert the end user there is something wrong (so they know, for example, that data is not refreshing in their charts)?

There are a few ways to do this, but most will involve developing what is called a rules engine. This example will help you build a basic rules engine, which can then be used to store errors in streams or send alerts to business users. There will be fewer pictures in this example as it builds off of the previous examples, which are meant to be done first and will teach many of the basic skills used here.

Tutorial Section 1: Add Smarter Logging

1. On **DemoThingRollUpDT**, click to modify the custom service called **UpdateAllDataTableEntries.**

2. Right now, this service has no error handling.

   a) The inner service (**CustomAddOrUpdate**) has a debug print statement for failures.

      i) It's likely that these errors will happen often, and we don't want to overcrowd the logs with errors every time they happen

   ii) These are still helpful, though, in case we want to know exactly which times failed and when for debugging the application and fixing the underlying issues

b) However, these errors don't help at all if the thing does not exist on the Foundation server

   i) If the thing doesn't exist, an exception will be thrown when the properties are accessed in the inner service call

   ii) This means that the service will fail, but no one on the application-side will know why

   iii) Putting both statements into the try-catch will allow for smarter logging.

c) Adjust your code to add a try-catch block, so that it looks like this:

```
// This returns faster than QueryImplementingThingsWithData, and allows us to still
// retrieve the data off every
// thing with no arbitrary, hard-coded limits. Doing it this way instead of looping
// through the table ensures newly created things get added to the table
var allThings = ThingTemplates["DemoTT"].QueryImplementingThings({maxItems:
      Things["DemoAppUtility"].MaxItems});

var counter = 0;
for(var i = 0; i < allThings.rows.length; i++) {
    var thingName = allThings.getRow(i).name;
    try {
         var thing = Things[thingName];
         var success = me.CustomAddOrUpdate({DateIn: thing.LastServiceDate, NumberIn:
             thing.NumberProp, ID: thing.ID, StringIn: thing.StringProp, NameIn:
             thing.name});
        if(success)
           counter++;
        else
            throw "Thing existed, properties could be found, but still entry was"
                 + "not updated in DT.";
    } catch(err) {
        logger.error(me.name + ": UpdateAllDataTableEntries: Could not update "
            + thingName + "; Message: " + err.toString());

        //(OPTIONAL TODO) Could also log this issue to a stream, which can then be
        // displayed and queried on a mashup
        // Note: ensure any errors occurring here are not caused by permission issues
        // and that the user has access to property read on the thing template level
         // Also note: this is not atomic; if one row fails, the rest will still update
    }
}

var result = counter; // Represents the number of things added or updated successfully
```

d) Notice how the error message is printing, with the name of the entity calling the service, followed by the name of the service itself, followed by the actual exception

   i) This ensures errors are searchable in the logs, and can be traced back to their causes easily

   ii) Likewise, printing the actual exception like this ensures that the error is not "swallowed" by our code, printing something which makes no sense or isn't really related in the logs

iii) This is error message best practice (see the appendix)

e) One last important note on database transactions in ThingWorx:

i) ThingWorx will not be aware of any issues if the outer service does not throw an exception from within the catch of the inner service, and so will not roll back any database commits

ii) This can result in duplicate entries if a unique ID field or key is not used for the add or update, which is why in this example, the ID field is a unique property on each thing

iii) For atomic updates, where either everything is updated or nothing is, or in the case where updates center around field values, but not unique identifiers, then the catch should always throw an exception so that the Foundation server and database are in consistent states

iv) So, our service here uses a try-catch within a loop to ensure that if one row fails, the rest still update. Logging this failure to the logs is good (what we do here), though storing it in a stream that can be reviewed by administrative-type end users (who don't have backend server access, but still want to know what issues there are and why) would be better

End Section 1: Review

In this section we wrote to the logs a more specific error message, something better for communicating the problem to the server admins. We also identified the proper location to insert the error into a stream. If we wanted to go all the way with this error handling, then we would create a mashup for administrative users of the application to view, which presents the error stream in a grid, along with a button for acknowledging errors. Then we could configure whether the alarm appears acknowledged by using state-based formatting on the status field.

This would enable any user of the application, whether they have a technical background or not, to review and potentially resolve issues on their own, especially if the issue is caused by data entry on another mashup, where things can be created or deleted by end users. This is especially beneficial for end customers of partners, who often resell the application to less technically experienced users.

Tutorial Section 2: A Basic Rules Engine (Create Alarms)

1. Create a new data shape for alarm information called **AlarmNotificationDataShape** with 8 fields:

a) **AlarmName** – STRING

b) **AlarmDescription** – STRING

c) **AlarmPriority** – STRING

d) **AlarmTimestamp** – DATETIME

e) **NotificationStatus** – STRING

f) **Notes** – STRING

g) **GroupToNotify** – STRING

h) **NotifyManagers** – BOOLEAN

i) Don't forget to tag the entity and add it to the right project (these entities will be provided for download and import down below, as some of them are quite complex, and as such, they have the same **DemoProject**, but an additional tag: "**Applications:RulesEngine**")

2. Create two new streams called **DemoTT_AlarmStream** and **DemoTT_EventStream** to store and allow triggered alarms and events to be reviewed

   a) Use **AlarmNotificationDataShape** for the data shape for both of these

   b) Don't forget to tag these and add them to the right project once again

   c) Note: these are named this way for a reason; in the provided Rules Engine code (steps 4b and 4c below), the name of the alarm and event streams are derived from the thing template name, where it is assumed that each thing template will correspond to a different region or type of device, etc. An alarm is anything which has a configured rule, whether notifications are enabled or not for that rule. An event is any other trigger (an alarm with "EVENT" for NotificationStatus)

3. Next, we need a data table to store alarm rules, called **AlarmDataTable**

   a) This will be used to store the rules themselves, which describe the frequency an alarm must be triggered before a notification is sent, e.g. 8 times in 1 hour, or every day for a week, etc.

   b) This is a complicated entity with services designed to function on a mashup discussed in the next section. Therefore, this and other supporting entities have been provided

   c) Import the provided *RulesEngine.xml* file to get these entities, all tagged with the following: **Applications:RulesEngine** (this should not override any existing entities)

   d) The relevant services on the **AlarmDataTable** are referenced throughout the coming steps

4. Create a new thing shape called **RulesEngineThingShape** and be sure to tag it and set the project

   a) This thing shape will have all of the services and subscriptions required to generate alarms and events and send notifications to end users

   b) In the next example, a much more complicated version of this is provided to demonstrate a more complete application. The idea here is that each thing shape represents a different type of device while each thing template represents a different region for devices

5. Modify the **RulesEngineThingShape** so that alarms can be checked and triggered

   a) Add a new service called **CheckForAlarmNotification**

      i) This should take in two inputs:

         (1) **AlarmName**– STRING – required

(2) **AlarmTimestamp** – DATETIME – not required

ii) This should return an INFOTABLE; ds: **AlarmNotificationDataShape**

iii) Here is the code:

```
logger.trace(me.name + ": Entering CheckForAlarmNotification... ");

var flag = false;
var time;
var notificationStatus = "";

// Have to initialize the result here so we can set it within each if-else scope below, and we
need an infotable to convert the JSON object to a valid return type.
var result = "", resultIT =
Resources["InfoTableFunctions"].CreateInfoTableFromDataShape({infoTableName: "InfoTable",
dataShapeName: "AlarmNotificationDataShape"});

if(AlarmTimestamp === null || AlarmTimestamp === undefined)
        time = Date.now();
else
        time = AlarmTimestamp;

var alarm_it = Things["AlarmDataTable"].GetAlarmInfo({AlarmName: AlarmName});
// We want to reduce the number of total queries, so here is where we convert the AlarmName into
// an actual alarm infotable with all its fields populated. If there is nothing in this entry,
// as in the length is 0, then we know the alarm is not in the table at all (so return false).
var alarmFoundFlag = alarm_it.length>0;
logger.trace(me.name + ": CheckForAlarmNotification: Alarm Info Found for " + AlarmName + "? "
                                                                            + alarmFoundFlag);
if(alarm_it.length > 0) {
    for(var a = 0; a < alarm_it.length; a++) {
        // This loop enables multiple notification rules to be configured for the same alarm name
        var alarmInfo = alarm_it[a];
        notificationStatus = "ACKNOWLEDGED";

        // Have to create a fresh object and not pass in AlarmDetails directly, or the
        // info table doesn't capture the data for some reason
        var currentAlarm = new Object();
        currentAlarm.AlarmName = AlarmName;
        currentAlarm.AlarmDescription = alarmInfo.Description;
        currentAlarm.AlarmTimestamp = time;
        currentAlarm.AlarmPriority = alarmInfo.Priority;
        currentAlarm.NotificationStatus = notificationStatus;
        currentAlarm.GroupToNotify = alarmInfo.GroupToNotify;
        currentAlarm.NotifyManagers = alarmInfo.NotifyManagers;

        logger.trace(me.name + ": CheckForAlarmNotification: Alarm enabled "+alarmInfo.Enabled);
        if(alarmInfo.Enabled) { // Alarm is in the table, and it is enabled
            // Check if notification should be sent this time or not
            if(alarmInfo.Occurs > 0) {
                // Then there are additional rules to check
                var counter_o = 1; // Start at 1 to include the current alarm
                var counter_cd = 0;

                // Increment down, and keep track of where we last were
                var leftOff = me.AlarmTable.length-1;

                do {
                    // Which day it is based on consecutive days
                    var cd_threshold = time-counter_cd*24*60*60*1000;

                    // First, calculate the threshold based on the units and consecutive day
                    var threshold = 0;

                    if(alarmInfo.Units === "Days")
                        threshold = cd_threshold-alarmInfo.Every*24*60*60*1000;
                    else if(alarmInfo.Units === "Hours")
                        threshold = cd_threshold-alarmInfo.Every*60*60*1000;
                    else if(alarmInfo.Units === "Years")
                        threshold = cd_threshold-alarmInfo.Every*365*24*60*60*1000;

                    // Then increment backwards through the alarm table until the threshold
                    // is hit to look for occurrences
                    for(var i = leftOff; i >= 0; i--) {
                        var row = me.AlarmTable.getRow(i);

                        // We only care about the entries that have the same name as the
                        // current alarm, but instead of querying for them, go through the
                        // main AlarmTable anyway so old entries can be purged in one loop
                        if(row.AlarmName === AlarmName) {
                            // The row datetime goes down in ms. When it equals the threshold
                            // (in ms) or is lower than it, then we increment the cd counter

                            if(time <= threshold) {
                                if (counter_cd < alarmInfo.ConsecutiveDays) {
                                    // If there are more days to go, reset to start counting
                                    // occurrences over for the next cd
                                    counter_o = 0;

                                    // But remember where we left off
                                    leftOff = i;

                                    // This will break the loop completely if it is the last
                                    // time, or reset for the next consecutive day otherwise
                                    counter_cd++;
                                    break;
                                } else {
                                    // Delete all remaining entries as they are outdated
                                    logger.debug(me.name + ": CheckForAlarmNotification: Found
                                                              and removed old entry in AlarmTable
                                                              for AlarmName " + row.AlarmName);
                                    me.AlarmTable.Delete(row);
                                }
                            } else {
                                // Occurrence for this day is detected; increment o counter
                                counter_o++;

                                // If there are no older entries in the table, then this is
                                // it for the loop, and the above timestamp check will never
                                // hit. So, here we have to update the flag if the counter
                                // has incremented enough so that the alarm will notify
                                if(counter_o >= alarmInfo.Occurs
                                                   && counter_cd === alarmInfo.ConsecutiveDays)
                                    flag = true;
                            }
                        }
                    }
                } while(counter_cd < alarmInfo.ConsecutiveDays);

                if(!flag) {
                    // If after all this, the conditions to fire the alert are not met, then no
                    // alarm is needed.
                    notificationStatus = "NO NOTIFICATION NEEDED";
                } else {
                    // Flag was true, so notification will be sent
                    notificationStatus = "NOTIFICATION PENDING";
                }
            } else {
                // If Occurs is 0, notify every time
                notificationStatus = "NOTIFICATION PENDING";
            }
        } else {
            // If Alarm is in the table, but not enabled
            notificationStatus = "NOTIFICATIONS DISABLED";
        }

        // Finally, add the current alarm to the alarm table
        currentAlarm.NotificationStatus = notificationStatus;
        // Don't need to add the alarm if it isn't enabled
        if(notificationStatus !== "NOTIFICATIONS DISABLED")
```

```
            me.AlarmTable.AddRow(currentAlarm);

        resultIT.AddRow(currentAlarm);

    }

    // And return the updated alarm details, which we only do once, so outside of the a loop
    result = resultIT;
} else {
    // Alarm is not in the table, so it will be added to the event stream
    var currentEvent = new Object();
    currentEvent.AlarmName = AlarmName;
    currentEvent.AlarmTimestamp = time;
    currentEvent.NotificationStatus = "EVENT";

    resultIT.AddRow(currentEvent);
    result = resultIT;

}
```

b) Add a new service called **AddAlarmToStream**

i) One input: **AlarmInfo** – INFOTABLE ds: **AlarmNotificationDataShape** – Required

ii) There is no return for this service

iii) Notice that in the source field, both the name of this thing and the name of the alarm are given. This allows for queries to make use of the indexed "source" field, resulting in faster queries with better performance on mashups (see appendix for details on best practice)

iv) Here is the code:

```
logger.trace(me.name + ": Entering AddAlarmToStream...");
// This is in accordance with the DemoAppUtility create userbase naming conventions:
var alarmStreamSuffix = "_AlarmStream";
var streamName = me.GetThingTemplate().name + alarmStreamSuffix;

// This field is set to null for unacknowledged alarms, which are acknowledged
// when someone adds notes manually from a mashup
AlarmInfo.Notes = "";

// Just use now as default if nothing is given since it is required for some reason
if(AlarmInfo.AlarmTimestamp === null || AlarmInfo.AlarmTimestamp === undefined)
    AlarmInfo.AlarmTimestamp = Date.now();


var params = {
    values: AlarmInfo,
    source: me.name,
    timestamp: AlarmInfo.AlarmTimestamp
};

try {
    Things[streamName].AddStreamEntry(params);
    logger.trace(me.name + ":AddAlarmToStream: Added with source field "
                  + params.source);
} catch(err) {
    logger.warn(me.name + ": AddAlarmToStream: Could not add to alarm stream entry
                 with source:" + params.source + "; Message: " + err.ToString());
}
```

c) Add a new service called **AddEventToStream**

i) One input: **AlarmInfo** – INFOTABLE ds: **AlarmNotificationDataShape** – Required

ii) There is no return for this service

iii) Notice the same in the source field here, for the same reasons

iv) The code is a little different this time, but not much:

```
logger.trace(me.name + ": Entering AddEventToStream...");
// This is in accordance with the DemoAppUtility create userbase naming conventions:
var eventStreamSuffix = "_EventStream";
var streamName = me.GetThingTemplate().name + eventStreamSuffix;

// Just use now as default if nothing is given since it is required for some reason
if(AlarmInfo.AlarmTimestamp === null || AlarmInfo.AlarmTimestamp === undefined)
    AlarmInfo.AlarmTimestamp = Date.now();

var params = {
    values: AlarmInfo,
    source: me.name + ": " + AlarmInfo.AlarmName,
    timestamp: AlarmInfo.AlarmTimestamp
};

try {
    Things[streamName].AddStreamEntry(params);
    logger.trace(me.name + ":AddEventToStream: Added with source field "
                 + params.source);
} catch(err) {
    logger.warn(me.name + ": AddEventToStream: Could not add to event stream entry with
                source:" + params.source + "; Message: " + err.ToString());
}
```

d) Add an info table property called **AlarmTable** with the **AlarmNotificationDataShape**

i) It should be marked as persistent.

ii) This is normally something to avoid when creating info table properties, but in this case, the info table should never contain more than the number of rules times the number of occurrences times the number of days. Since the number of occurrences will usually be a small number (let's say 5), and the number of days will usually be a small number (let's say 3), and the number of rules is kept to something like 20 total, then that is 20*5*3 = 300 entries maximum, and that is assuming that every rule uses the number of occurrences over many days (see appendix for details on best practices here).

Best Practice Note:
An info table property can typically handle on the order of hundreds of rows without too much of a performance hit, though it is worth it to consider how this many numbers of rows times the total number of things will affect performance, and to ensure the Foundation server has enough memory allocated to handle the load.

e) Add a service called **PurgeAlarmTable**

i) This should take one input: **AlarmName** – STRING – required

ii) There is no return for this service.

iii) Here is the code:

```
var size = me.AlarmTable.length;


// Increment through the AlarmTable backwards and remove anything which matches the
// given alarm name
for(var i = size-1; i >= 0; i--) {
    var row = me.AlarmTable.getRow(i);

    if(row.AlarmName === AlarmName)
  me.AlarmTable.RemoveRow(i);
```

```
            }
```

f) Add an event called **TriggerAlarm** using **AlarmNotificationDataShape**, though the only field we care about here is **AlarmName**

   i) This alarm is triggered by the data change events (which we will add below)

   ii) The data change event subscription doesn't do this code directly as there may be other triggers for the same alarm throughout the application, including calls from the Edge itself, so that's why we put this event here, to create a more versatile application.

g) Create a subscription to **TriggerAlarm**

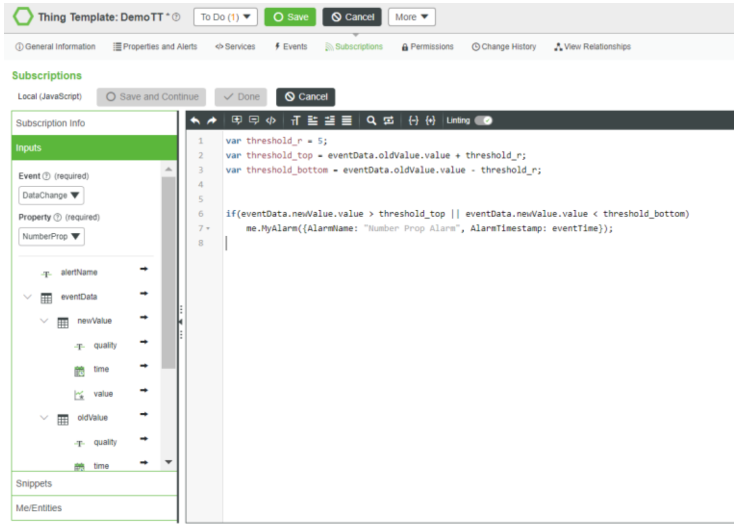
   i) It should appear as an option in the Event drop-down box on the Inputs tab.

   ii) Give the alarm a name on the Subscription Info tab, which is also where you enable it.

   iii) Here is the code:

```
logger.trace(me.name + ": Entering TriggerAlarmSubscription...");

var updatedAlarmDetails = me.CheckForAlarmNotification({AlarmTimestamp:
                              eventData.AlarmTimestamp, AlarmName: eventData.AlarmName});
logger.trace(me.name + " TriggerAlarmSubscription: Check for Alarm Notification Result
                              has " + updatedAlarmDetails.length + " row(s).");

for(var a = 0; a < updatedAlarmDetails.length; a++) {
    // We need to copy the row into an infotable to send it into the
    // stream update services below
    var alarmDetailsIT = Resources["InfoTableFunctions"].CreateInfoTableFromDataShape(
                      {infoTableName: "InfoTable",
                      dataShapeName: "AlarmNotificationDataShape"});

    var alarmDetailsRow = updatedAlarmDetails.getRow(a);
    logger.debug(me.name + " TriggerAlarmSubscription: Alarm Name " +
               alarmDetailsRow.AlarmName + "; Notification Status: " +
               alarmDetailsRow.NotificationStatus);

    // If the alarm requires a notification
    if(alarmDetailsRow.NotificationStatus === "NOTIFICATION PENDING") {
        alarmDetailsRow.NotificationStatus = "NOTIFYING";
        var group = "";
        var notifyManagers = false;

        // Determine group to notify and if managers need notification as well:
        if(eventData.GroupToNotify !== null && eventData.GroupToNotify !== undefined)
            group = eventData.GroupToNotify;
        else
            group = alarmDetailsRow.GroupToNotify;

       if(group === "Regional") // Then parse thing template name per DemoAppUtility
            group = me.GetThingTemplate().name + "_Group"; // naming conventions

        if(eventData.NotifyManagers !== null && eventData.NotifyManagers !== undefined)
            notifyManagers = eventData.NotifyManagers;
        else
            notifyManagers = alarmDetailsRow.NotifyManagers;

        // Then create the email
        var params = {
            subject: alarmDetailsRow.AlarmName + " alarm fired on " +
               me.name + "; Priority: " + alarmDetailsRow.AlarmPriority,
            body: "Asset Name: " + me.name + "; " + "<br><br>Alarm Description: " +
               alarmDetailsRow.AlarmDescription + "<br><br>Alarm Time: " +
               alarmDetailsRow.AlarmTimestamp,
            GroupToNotify: group,
            NotifyManagers: notifyManagers
        };

        // And send it (we will add this service below, after importing the mail
        // extension, so to test in the meantime, comment this out)
        alarmDetailsRow.NotificationStatus =
       Things["AlarmNotifier"].SendEmailForAlarm(params);
        logger.debug(me.name + ": TriggerAlarmSubscription notification status after
               send: " + alarmDetailsRow.NotificationStatus);

        // Then remove the previous alarms from the AlarmTable to reset for next time
        me.PurgeAlarmTable({AlarmName: alarmDetailsRow.AlarmName});

        // Finally, add the alarm with the results of the send attempt to alarm stream.
        // We do this in the loop because we don't want to add every single alert, only
        // those alerts which notify users because the rules are satisfied (alarms).
        // Have to copy it into an info table to be able to pass it as a parameter below
       alarmDetailsIT.AddRow(alarmDetailsRow);
        me.AddAlarmToStream({AlarmInfo: alarmDetailsIT});
    } else if(alarmDetailsRow.NotificationStatus === "EVENT") {
        // Have to copy it into an info table to be able to pass it as a parameter below
       alarmDetailsIT.AddRow(alarmDetailsRow);
        me.AddEventToStream({AlarmInfo: alarmDetailsIT});
    } else if(alarmDetailsRow.NotificationStatus === "NOTIFICATIONS DISABLED") {
        // We still might want to review alarms that occur, even if no notifications
        // Have to copy it into an info table to be able to pass it as a parameter below
       alarmDetailsIT.AddRow(alarmDetailsRow);
        me.AddAlarmToStream({AlarmInfo: alarmDetailsIT});
    } else {
        logger.warn(me.name + ": TriggerAlarmSubscription: Unknown Notification Status
               after CheckForAlarmNotification service call: " +
               alarmDetailsRow.NotificationStatus);
    }
}
logger.trace(me.name + ": Leaving TriggerAlarmSubscription.");
```

h) Lastly, trigger the **TriggerAlarm** event in response to property updates



Event (required): DataChange
Property (required): NumberProp

```
var threshold_r = 5;
var threshold_top = eventData.oldValue.value + threshold_r;
var threshold_bottom = eventData.oldValue.value - threshold_r;


if(eventData.newValue.value > threshold_top || eventData.newValue.value < threshold_bottom)
    me.MyAlarm({AlarmName: "Number Prop Alarm", AlarmTimestamp: eventTime});
```

i) Move the **NumberProp** and **StringProp** properties to the thing shape so we can subscribe to their data change events

ii) Use an arbitrary threshold for the number property, subscribing to its data change event as seen in this image:

iii) Here is the code:

```
var threshold_r = 5;
var threshold_top = eventData.oldValue.value + threshold_r;
var threshold_bottom = eventData.oldValue.value - threshold_r;

if(eventData.newValue.value > threshold_top || eventData.newValue.value <
        threshold_bottom)
    me.TriggerAlarm({AlarmName: "Number Prop Alarm", AlarmTimestamp: eventTime});
```

6. We will test this below, after adding one more service (which is referenced in the subscription **SendEmailForAlarm**) on a **MailThing** that we haven't yet created for sending emails.

End Section 2: Review

In this section, we learned how to create a data shape for an event, which we also used for a stream. Then we created the rules engine service, which has comments to step through the process of verifying if an alarm needs to be sent or not. We added the service to update the alarm and event streams, as well as adding to the alarm info table property for keeping track of how many times an event has happened in the near past. As mentioned above, we can use the info table type data storage for this reason because the table is never allowed to grow too large, and instant updates and retrievals of data are needed, so there is no other choice (see appendix for details).

The purge happens in two places: within the check for the alarm, to see if it needs to send a notification and for any entries too old to matter as of right now; and within the notification service, to remove recent entries after the notification is sent, so the count can start again. We added the purge service above, but there is no mechanism implemented yet which reduces the amount of spam sent overall. Right now, if the issue happens 3 times in a minute, then an email will be sent out every minute. An additional query against the streams themselves could be used for advanced customization like this.

Finally, we created the event trigger for the data change event and demonstrated how to trigger the alarm, and subsequently, send a notification. We don't send the notification in the property subscription itself, as that would require us to repeat the same code in many subscriptions to many different properties. Instead, we subscribe to our created event, and then trigger this from wherever we would like. Here we use a data change event subscription, but we could just as easily trigger the event from the Edge device in response to its listening there, or from anywhere else which fits the use case. The code for the notification is repeated just once, and therefore, it is easier to maintain.

Tutorial Section 3: Configure the Mailer, Add the Alarms, and Test

1. Now, create the notification mechanism by importing the Email Extension

    a) First, find and download the Mail Extension, following the steps in the Help Center

    b) Once you have successfully imported the latest version of the Email Extension, create a new thing using the **MailServer** thing template (called "**AlarmNotifier**" here)

        i) Note that there is additional configuration required for importing entities in the newer versions of ThingWorx, involving updates to the "*platform-settings.json*" file

        ii) For details on how to enable extension imports, see the Help Center

    c) Next, configure the **AlarmNotifier**

        i) Use the **MailServer** thing template which was added in the import

        ii) For Gmail, use this information on the Configuration tab:

            (1) SMTP server: **SMTP.gmail.com**

            (2) SMTP Server Port: **465**

(3) POP3 Server: **POP.gmail.com**

(4) POP3 Server Port: **995**

(5) Use TLS: **not checked**

(6) Use SSL: **checked**

iii) You must provide a valid email address and account password for ThingWorx to be able to send emails on behalf of that account (and there may be requirements on the email server side as well, see KCS Article CS200624 for details regarding Gmail)

iv) An admin or "DO_NOT_REPLY" address is recommended for use, though what appears in the "sender" field can be modified independently in the service itself

d) Once configured, add a service called **SendEmailForAlarm** to the **AlarmNotifier** thing to send the email

i) Take in 4 parameters: **subject, body,** and **GroupToNotify** (STRING, required) and **NotifyManagers** (BOOLEAN, not required)

ii) Set the output to type STRING

iii) Under the "me" tab, find the **SendMessage** snippet

iv) Fill out the "from" field with the desired sender display name

v) Fill out the "subject" and "body" fields with the input parameters

vi) For now, hard code the "to" field for an address you can check

vii) Eventually, you will want to determine which users should be sent which alerts, perhaps having a separate data table to store which alerts go to which groups, groups which can then contain specific users (whose email addresses can be obtained for use in this service)

e) Here is the full code of the **SendEmailForAlarm** service:

```
if(GroupToNotify === null || GroupToNotify === undefined || GroupToNotify === "" ||
                                                         GroupToNotify === "Regional") {
   // Need to replace "Regional" with the group name before entering this service, which
   // doesn't know which thing template is being used, and therefore which group to use
   GroupToNotify = "Administrators";
    body = "NOTICE: NO GROUP TO NOTIFY SPECIFIED. Sent to Administrators. " + body;
}

var usersToNotify = Groups[GroupToNotify].GetGroupMembers();

// To notify managers, the group naming convention goes like "templateName_AdminGroup"
// We know the group name, though, so we have to do some string modifications here
if(NotifyManagers && GroupToNotify !== "Administrators") {
    var templateName = GroupToNotify.split("_")[0];
    var adminGroupName = templateName + "_AdminGroup";

    var adminsToNotify = Groups[adminGroupName].GetGroupMembers();
    usersToNotify = Resources["InfoTableFunctions"].Union({t1: usersToNotify,
                                                           t2:adminsToNotify});
}

var failures = "";
```

```
for(var a = 0; a < usersToNotify.rows.length; a++) {
    var username = usersToNotify.GetRow(a);

    var params = {
        subject: subject,
        from: "DO_NOT_REPLY@ptc.com",
        to: Users[username].emailAddress,
        body: body
    };

    try {
       me.SendMessage(params);
    } catch(err) {
        failures += username + "; ";
    }
}

if(failures !== "")
    result = "Failed to send to: " + failures;
else
    result = "Everyone notified successfully.";
```

2. Now you will want to create a mashup to enable adding the alarms from within the application, and reviewing the alarms and responding to them once they are triggered

   a) Since this would be complicated to describe in detail, a mashup has been provided in the import from the previous section (called **NotificationRulesMashup**)

   b) On the provided mashup, select the "New" checkbox before typing

      i) Note that this will clear what has been typed in the other boxes, so check this before entering the new name and description

      ii) This box must be checked because the row in the grid cannot easily be deselected at present

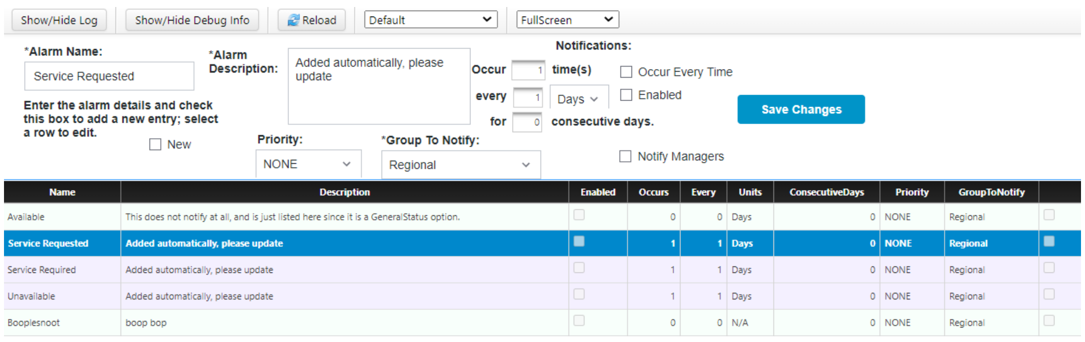
      iii) Whatever row is selected will be overwritten if the "New" checkbox isn't checked

   c) From the import in the previous section, find the mashup called **NotificationRulesMashup**, or create the rules manually by adding rows to **AlarmDataTable**

   d) Note that the name for the alarm needed here (based on the code in step 5h in section 2 above) is "**Number Prop Alarm**", spelled exactly this way, with caps and all

   e) If the alarm is not listed in the table, it will be considered an event, so to create events instead of alarms, use any AlarmName not listed in the table in the data change event subscriptions

   f) If you plan to create the mashup manually for practice, see the screenshot shown here for details about what features will be required:



*Alarm Name: Service Requested

*Alarm Description: Added automatically, please update

Enter the alarm details and check this box to add a new entry; select a row to edit. ☐ New

Priority: NONE

*Group To Notify: Regional

Notifications:

Occur 1 time(s)

every 1 Days

for 0 consecutive days.

☐ Occur Every Time

☐ Enabled

☐ Notify Managers

Name	Description	Enabled	Occurs	Every	Units	ConsecutiveDays	Priority	GroupToNotify	
Available	This does not notify at all, and is just listed here since it is a GeneralStatus option.	☐	0	0	Days	0	NONE	Regional	☐
Service Requested	Added automatically, please update	☐	1	1	Days	0	NONE	Regional	☐
Service Required	Added automatically, please update	☐	1	1	Days	0	NONE	Regional	☐
Unavailable	Added automatically, please update	☐	1	1	Days	0	NONE	Regional	☐
Booplesnoot	boop bop	☐	0	0	N/A	0	NONE	Regional	☐

3. Since there is a timer to update the **NumberProp** with random numbers, this alarm should hit frequently (this was created in Example 2, Section 3)

   a) Once the alarm is added to the data table as shown in the previous step, then the rules engine should be working

      i) You should delete all but a handful of the **DemoThings** before turning on the alarm, as it will trigger emails be sent for each thing

      ii) This means that if there are 1000 **DemoThings**, as many as 1000 emails may be received at one time; test this out with only a handful of **DemoThings**

   b) Check that the **AlarmTable** property on a thing implementing the **DemoTT** thing template is updating, and that the emails are being received

   c) Try different rules to ensure everything works as expected

   d) To manually fire the alert events, change the value of the **NumberProp** on a given **DemoTT** type thing to any number larger or smaller than 5 plus the previous value

   e) Create a mashup called **AlarmReviewMashup** where the stream entries from the **DemoTT_AlarmStream** can be reviewed; create one for events as well or use the same, or check out Appendix III for a Coffee Machine Demo App that includes this plus other advanced features.

End Section 3: Review

In this section, we learned how to configure the mailer with Gmail account information and added the service for sending the email to the end user. Notice how we pull the group to notify from the data table that stores the rules, so that administrators of the application can configure which groups get which alerts manually from the provided mashup. Likewise, multiple entries with the same alarm name can be added to the **AlarmDataTable** to enable the same notification going to multiple groups.

At this point, the application is complete from end to end. There is a thing shape with all of the notification and rules logic, there is a mashup for adding rules and a data table to store them. There is a timer which updates all of the **DemoThings** for simulation purposes but making these into actual remote things is simple: use the **RemoteThing** template, apply the same thing shapes, and send the property updates from the Edge. Don't forget to use the **DemoAppUtility** thing to generate users and user permissions (modify the service **InitializeUserbaseAndPermissions** for new templates and mashups).

Example Completion – Summary:

This concludes the third example in this document. You should now know how to:

- Create data shapes for events
- Create streams to store past events
- Send emails when events occur
- Subscribe to data change events
- Build a basic rules engine

Appendix I: Definition of Terms

[1] Mashups – Essentially webpages to which end users (employees or customers, depending on the use case) can navigate in order to review information about the assets and use the application.

[2] IoT Application – Any application built primarily for machine-to-machine, or thing-to-thing, interaction, where machines can notify other machines of their statuses, order stock on their own, schedule maintenance, etc., with very limited human interaction required for many features. Humans often use the application only to review things are working as expected.

[3] Scalability – The ability of an application to go from a very small, demo-like environment (called Proof of Concept, or PoC) to a very large, fully live environment (called Go Live, or Production). Total number of things desired should be considered during the design phase, and never overlooked in order to just "get something working".

[4] Maintainability – The ability of an application to be upgraded over time, and improved with new features, including the plan for upgrades and improvements, which should be thought out and considered during the initial design phase.

[5] KCS – PTC's Searchable Knowledgebase and first stop shop for all things documentation. Look for the box which says "Search the Knowledgebase" on the PTC Support Home Page.

[6] Edge devices – Collection of assets, products or devices of some kind which are then sold or rented out to end customers. These are the fleet of devices which sit out there in the world somewhere, hooking up to the Foundation via the internet.

[7] Nodes – In a Network, nodes are any device which connects to other devices, which would therefore include both the fleet of Edge devices, the Foundation, and any other components which utilize the Foundation server (like external applications).

[8] Controller – In the traditional MVC model for web applications, the Controller is what retrieves data from the Model and sends it to the View, and vice versa. ThingWorx contains the potential to fulfill all three components, with the mashups being the View, the Foundation server itself being the Controller, and the embedded or accompanying database being the Model. Edge devices hook up to the Foundation, the Controller, which then siphons data from each device into the Model (the database), so it can later be retrieved on the View (the mashups). For this reason, ThingWorx is said to be a "service provider" at times, when other external components are used, for example when Windchill is the "resource provider", taking on the role of the Model while the Model built into ThingWorx goes unused.

[9] Javascript functions – Javascript (or JS) is a web language which is very versatile and easy to use. Javascript functions are stored as services in ThingWorx, allowing for developers to build applications very simply and without much experience.

[10] Repositories – The location on the Foundation server where files are stored (there can be many repositories).

[11] OOTB – Stands for "Out of the Box", meaning that the functionality is built-in and should work with little extra effort.

[12] REST API – REST is a protocol (like a language) used to structure messages sent across the internet. It is very common for web applications, typically being used to request information from one node to another. ThingWorx has a REST API built-in, meaning there are a bunch of commands which can be used to return information independent of the use of a browser, allowing for easier integration of 3rd party apps which just need to pull data from ThingWorx.

[13] Info Tables – An info table is a JSON object, which just refers to the structure of the message. In ThingWorx, info tables are used to keep track of temporary information, often being the return result of services. They can then be iterated through (like a list) and manipulated in some way before being returned again, or parsed and stored in memory somewhere.

[14] Deadlocks – This occurs when information in a database is "locked" to other processes because of updates, but the lock won't release until one of the other processes completes (which it can't do, because it needs the lock to open); a catch 22.

[15] Race Conditions – When information is updated from multiple locations in an application, resulting in incorrect data being stored or displayed, and sometimes even deadlocks depending on the design.

[16] Cost – In Software, the cost of an operation refers to the length of time that operation takes to complete.

[17] Streams (or Value Streams) – Data structures designed for time series data, which typically update from data ingestion (value streams) or from processes which then aggregate the ingested data to make it consumable on mashups (streams).

[18] Data Tables – Data structures which are very much like classical database tables, for information which will be updated frequently. It is important to ensure the number of things which update the data table at once are limited (or it will deadlock).

[19] Load Testing – An important step in ensuring an application is ready to go from PoC to Production. Many devices are created virtually to test how the Foundation will look when many things are connected at once, and data is created which perfectly mimics the type of data found in Production to prove the application handles real data correctly (done in Sandbox).

[20] Production – The server on which the public-facing application runs, i.e. the one which customers or employees use.

[21] JVM Memory – JVM stands for "Java Virtual Machine", the component of the ThingWorx Foundation server which actually runs all of the services and allocates the memory required for the application to function. This has a limited amount of memory, as configured in the Tomcat Java Options (see the installation guide for details), which should be considered precious.

[22] Session Variables – These are allocations of memory that the JVM uses to keep track of information throughout a particular user's session (i.e. the time when they are logged into the Foundation). These take up memory on both the server and the end user side, and so should be considered expensive.

[23] Developer Tools – This is an application built-in to most browsers which allows for advanced debugging and performance analysis techniques, as well as assistance with mashup style and layout development, though that is not discussed here.

[24] Network Requests – These are calls from webpages (mashups) to the various parts of a web application (the Foundation) requesting specific information for display. Binding data to a widget will result in a network request upon mashup load.

[25] Query Microservice – This is used to offload the ThingWorx server by allowing query execution to occur in a separate process on the same or on a different physical machine. Query Microservice is recommended when the system uses queries that load and retrieve a very large amount of data in-memory, resulting in slowness or system crashes or when the system experiences slowness and unresponsiveness due to a large volume of queries executed as part of processing (even if the queries themselves are actually small, the load can add up). Read more about how to use this feature in the Help Center.

Appendix II: Quick Tip Chart and References

These topics are listed in the order they are referred to in the document above, with some exceptions where things are mentioned multiple times. The last entry is a link to the ThingWorx Best Practice Hub, an excellent resource for all topics not covered here.

Topic	Best Practice Note	Documentation
Ghost Entities	Call the services EnableThing and RestartThing after creating a new thing, and ensure that if an error has occurred, the thing is deleted using DeleteThing before the service execution completes. There is an extension to delete ghost entities in case they do occur, but this should prevent them from happening.	KCS Article – ghost entities Marketplace Extension
Session Variables vs. Mashup Parameters	Session variables utilize both server-side and end-user-side memory, and so are very expensive and if used too much, can cause performance issues. Try to avoid info tables as session variables, as these can be quite large at times. Mashup parameters store data only on the end-user-side, and so do not have the same potential to interfere with server stability.	KCS Article – sessions variable best practice with links using session and mashup parameters KCS Article – info tables and memory usage
Collection Widget	So long as the inner mashup makes use of mashup parameters and is relatively simple, this can be quite performant. Likewise, there are configuration options to improve load performance. Use this widget to build more advanced grids which contain charts and other things unable to be rendered in ordinary grids.	KCS Article – how to use collection widget Help Center – configuration options and properties
Responsive Mashups for Dynamic Web Pages	Using dynamic mashups as opposed to static enables the mashup to take up whatever size is available to the screen, improving how the mashup looks on a variety of screens. In Composer, there are options to view the mashup at different screen sizes for testing purposes. To ensure some things are always placed at the same point on the screen regardless of screen size, use a panel within a panel.	PTC Community Thread – build dynamic mashups PTC Community Thread – panels in panels and widget positioning
Advanced Grid	The Advanced Grid now comes standard with ThingWorx and has generally improved performance. The sort feature allows for a search bar to appear on the grid itself. When typing into this search bar, the information passed in is utilized immediately, and the result is a nice search aesthetic which can query against multiple columns at once. The sort must be manually constructed, though, so be sure to check out the Help Center.	Help Center

Updating Production Entities (Saving Thing Templates in Prod)	Never save Thing Templates in Production during regular business hours. When a Thing Template is saved, all Things implementing that Template must be restarted. If these are Edge devices, then the logs will be flooded with errors as these things fail to connect to their Foundation counterparts (while the restart is in process). This can result in a ton of errors to the logs, server instability, failures to reconnect, and loss of data (if there is no failover and things don't go well). The best method for updating production entities is to import the changes all at once by file or import from storage, outside of peak business hours (on a weekend, at night, etc.).	KCS Article – change management in ThingWorx
Java Runtime Memory (JVM) – Memory Usage	In Java, Garbage Collection (GC) is automatic and happens at set intervals. The ThingWorx Foundation is built on Java, and so the JVM manages when to clear out old memory blocks to make room for new ones. This can mean that the memory in the PlatformSubsystem will climb as things are deleted or data entries removed until the GC is triggered, and the old memory blocks removed. It is important to ensure the Foundation server has enough memory to run the essential functions and still grow in size during GC.	KCS Article – Java heap settings explained and overview of flags
Data Storage in ThingWorx	There are 4 primary ways to store data, and each with its own specific purpose in mind: info tables are for very transient data, being very memory intensive, and should be purged often; data tables are for permanent data stores which are updated often and from many locations; streams are for storing aggregated, historical data; and value streams are for storing data as it is ingested into the Foundation server. Purging mechanisms should be considered required, and data ingestion should be separated from data use on mashups as much as possible.	PTC Community Post – where should I store my TWX data? KCS Article – technical distinction (streams, VSs, DTs) KCS Article – info tables and memory usage KCS Article – when to choose which, data exporting KCS Article – tuning stream performance/stream processing KCS Article – foreign keys (DTs) KCS Article – purging data KCS Article – data aggregation
Influx DB	This is a persistence provider option (introduced in 8.4) which can be configured alongside the PostgreSQL database (which handles the property and entity persistence) to handle the data itself, the data table, value stream, and stream content. This data is then stored in an external database, designed specifically for ingesting and querying time-series data in real-time (with purge and aggregation support on the	Help Center – overview, installation, configuration details provided here PTC Community Post – walk through with pictures KCS Article – roadmap does include further capabilities of Influx within ThingWorx

	roadmap). Influx DB is not supported in Windows and is set up independently from the Foundation. Export functionality is currently not supported for Influx (in version 8.4).	
Timers and Schedulers	Whenever possible, do not subscribe to timers/schedulers on Thing Template (TT) level, and there should almost never be a reason to query in TT subscriptions even if they do prove necessary. Do subscribe to the timer event on the timer itself to avoid hitting the event processor as hard (the PlatformSubsystem is hit instead, and it is better equipped to handle the operation). If a query is needed, do it on the timer itself, and only once at the start of the subscription.	PTC Community Post – best practice and when to use
Edge Property Updates	It is almost always better to push updates from the Edge than to pull them from the Foundation server. This is because any communications from the Edge come from a designated communications subsystem, separate from the main memory pull. ThingWorx is always listening, so may as well put that memory to work!	KCS Article – property overview and configuration KCS Article – subscribed properties and remote property updates in TWX
Queries	The query parameter in any service which retrieves data in ThingWorx is applied only after the entire data set (based on the other parameters) is returned. For this reason, use the indexed datetime and source fields to return a limited amount of data (and improve performance). In Data Tables, there is the option to index an additional field (the primary key), as well as make use of the "id" field for much faster lookups. For massive queries (those which store a lot of data in-memory), or for fixing performance issues when there is a large volume of queries, consider using the external Query Microservice, introduced in 8.4.	Help Center – on queries Help Center – on data tables, including indexing fields Help Center – query microservice overview and installation details
Logging Error Messages	To ensure that all things which print to the logs come from an identifiable source, format error messages using a particular formula. This will save so many headaches later on, when someone on the team cannot remember which service possessed the logging statements (for debugging purposes) which are now spamming the logs. Irresponsible logging can prevent other issues from being resolved quickly at best, and at worst, cause the server to crash when the logs are bloated and the server memory used up. The right	KCS Article – how to use logging statements and view logs

	format is (where there is an exception err being caught): `logger.warn(me.name + ": serviceName: error message; Exception: " + err.toString())` Also, note that trace and debug statements will be the most common, only turned on and used when things go very wrong, followed by info statements (which will normally be visible), and warn and error, which should always be visible. A good service will print something if it is successful (if it is not run very often) and when something fails. Always print the actual error message, from the exception itself, to ensure that information isn't "swallowed" by the log. Also include the service name and thing name so that it can be traced back to its source quickly and efficiently.	
Smart Error Logging	Store errors into streams and use rules engines to keep track of sending notifications. Streams can be viewed and interacted with on mashups, providing the potential for an acknowledgement mashup, with accompanying email notification. This prevents administrators from needing to get into the logs to deal with application-based or expected issues. Be careful with using streams for this purpose, though, since updates are not immediate and depend on the stream processor to complete.	KCS Article – How to build a basic rules engine in ThingWorx (noting how AddStreamEntry does not add entries right away)
Data Change Events	Data Change events hit the event processing subsystem, which has fewer threads allocated to it than the platform or stream subsystems. Therefore, queries within data change events should be kept to a minimum to ensure that they complete quickly and release their threads. Deadlocks in these subscriptions can and will bring down the entire server. Limit the number of total times and the overall runtime of any queries used for data updates. Never use long remote service timeouts in data change events, and avoid remote service calls altogether.	KCS Article – how to do complicated event calculations PTC Community – performance design, pitfalls, troubleshooting
ThingWorx Best Practice Article Hub	This KCS Article is kept up-to-date by ThingWorx technical support with recent additions to the knowledgebase on the topic of best practice. Many topics are in this article which were not mentioned here.	KCS Article – ThingWorx Best Practice Hub

Appendix III: Coffee Machine Demo App

Within this download there are three entity files: *DemoApp.xml*, *RulesEngine.xml*, and *CoffeeMachineDemoApp.xml*. The first of these is simply all of the entities that are created in the tutorials provided here, save for the rules engine portion, which is found in the second download. Separating these two components this way allows for the customization of the demo app from the start, without any risk of overriding those changes when importing the logic for the rules engine later on.

The third of these files hasn't been referenced here yet, but it is a complete application designed to show the best practice for alarms and events tracking in a Smart and Connected Products scenario. The provided application is not designed to be used directly in production environments. It is instead designed to demonstrate one of the easiest ways to develop a notifications-based IoT application.

Each region in this coffee machine demo app should correspond to a thing template, and the type of device should be specified by the applied thing shape. In other words, **CoffeeMachineTS** replaces the **RulesEngineThingShape** on the thing template, but the thing template in this case should be named like "RegionX_DeviceTypeN" (so for the Coffee Machine App here, you might name one template "**Region1_CoffeeMachine01**"). The template also has to be tagged to be picked up by the roll up logic, with the default tag used here being "**Applications:CoffeeDemoApp**".

In a real application designed like this, you would typically have each thing template refer to a region or a factory, and then have different thing shapes with different properties and events as needed, but with similar handling for generating alarms and events. Usually, rules would be shared across all regions, and to send the emails to different groups based on region in this application, set the GroupToNotify to "Regional" (the default setting). The subscription to the triggered alarm, located on the thing shape, can then use the default naming conventions mentioned above. Note that the initialization services, especially for creating and assigning user permissions, within the **DemoAppUtility** thing will need to be modified manually to work with the new entities introduced here.

The **CoffeeMachineTS** has lots of services that were not created as a part of the tutorials. These are all used within the provided mashups. **IndividualCoffeeMachineMashup** simulates a cash register interface that would be used both by baristas using the machine, and by those looking at the machine's details and alarms and events for analysis and maintenance. Servicing can be requested or is considered required if parts are malfunctioning, and drinks can be ordered. Note that in order for this example to be self-contained, none of the properties are updated remotely here. That can be changed if a remote thing template is used, and the properties are updated by the Edge instead of upon button click when a drink is ordered. The order button can then just pull the property info from the Edge, or the property info can be pushed whenever it changes, however it makes the most sense in your application.

This mashup can be opened from the Administrator view: **OverviewCoffeeMachineMashup**. This mashup shows a roll up of all coffee machines with general status information and statistics, as well as whether or not they are connected and functioning correctly. Double clicking the grid opens a mashup with the alarms displayed, and entering a note there is how alarms are acknowledged in this application. The roll up logic runs on a timer also included with this application (**CoffeeMachineRollUpTimer**), updating the drink distribution, percent uptime, and longest service request info every 30 seconds. Clicking the drill down button will open the **IndividualCoffeeMachineMashup** for the selected thing.