WebSocket-based Edge MicroServer Developer's Guide

Version 5.3.2
June 2016
updated May 2017

Contents

About This Guide

The ThingWorx Edge WebSocket-based MicroServer (WS EMS) is used to provide a simple means for remote devices to connect quickly and securely to ThingWorx Core.

This document describes how to use the WS EMS.

Note

This document is accurate as of this release and is subject to change. For the latest documentation, see either the Reference Documents page for ThingWorx Edge products or the ThingWorx Edge Help Center, which are available through the PTC ThingWorx Support pages at https://support.ptc.com/appserver/cs/portal/.

Audience

This document is intended for programmers with at minimum a basic knowledge of JSON and the Lua scripting language. In addition to programming knowledge, you need to be familiar with ThingWorx Core, its concepts, and ThingWorx Composer.

Technical Support

Contact PTC Technical Support via the PTC Web site, phone, fax, or e-mail if you encounter problems using your product or the product documentation.

For complete details, refer to Contacting Technical Support in the *PTC Customer Service Guide*. This guide can be found under the Related Resources section of the PTC Web site at:

http://www.ptc.com/support/

The PTC Web site also provides a search facility for technical documentation of particular interest. To access this search facility, use the URL above and search the knowledge base.

You must have a Service Contract Number (SCN) before you can receive technical support. If you do not have an SCN, contact PTC Maintenance Department using the instructions found in your *PTC Customer Service Guide* under Contacting Your Maintenance Support Representative.

Documentation for PTC ThingWorx Products

You can access PTC ThingWorx documentation, using the following resources:

- PTC ThingWorx Help Centers — The PTC ThingWorx Core Help Center provides documentation for the PTC ThingWorx Core, including

ThingWorx Composer. The PTC ThingWorx Edge SDKs & WS EMS Help Center provides documentation for the ThingWorx Edge WebSocket-based MicroServer (WS EMS) and for the Edge SDKs. You can browse the entire documentation set in a Help Center, or use the search capability to perform a keyword search. To access the PTC ThingWorx Help Centers, visit PTC Help Center.

- PTC ThingWorx Reference Documentation — The Reference Documents page on the PTC Support site provides access to the PDF documents available for the PTC ThingWorx Edge SDKs and WS EMS, and the PTC ThingWorx Core at http://support.ptc.com/appserver/cs/doc/refdoc.jsp, These PDF documents include documents that explain system requirements. For ThingWorx Edge products, this document is entitled *ThingWorx Edge Requirements and Compatibility Matrix*.

  A Service Contract Number (SCN) is required to access the PTC documentation from the Reference Documents website. If you do not know your SCN, see "Preparing to contact TS" on the Processes tab of the PTC Customer Support Guide for information about how to locate it: http://support.ptc.com/appserver/support/csguide/csguide.jsp. When you enter a keyword in the Search Our Knowledge field on the PTC eSupport portal, your search results include both knowledge base articles and PDF guides.

Comments

PTC welcomes your suggestions and comments on our documentation. To submit your feedback, you can:

- Send an email to documentation@ptc.com. To help us more quickly address your concern, include the name of the PTC product and its release number with your comments. If your comments are about a specific help topic or book, include the title.
- Click the feedback icon in any PTC ThingWorx Help Center toolbar and complete the feedback form. The title of the help topic you were viewing when you clicked the icon is automatically included with your feedback.

1

Introducing the ThingWorx WS EMS

This section provides an overview of the ThingWorx Edge WebSocket-based MicroServer (WS EMS), explaining the relationship between the ThingWorx WS EMS and ThingWorx Core, and summarizes the purpose and key features of the ThingWorx WS EMS.

Components to Install

To connect and integrate with ThingWorx Core, you can install two separate software components remotely on edge devices:

- ThingWorx Edge WebSocket-based MicroServer (WS EMS) — The WS EMS is the communication conduit that provides a secure communication channel to ThingWorx Core. The WS EMS is a separate process, so it can support communications from one or more devices. In addition, the WS EMS provides a RESTful HTTP interface, allowing other applications to communicate with it. The WS EMS translates the REST APIs into AlwaysOn protocol messages that it then sends to ThingWorx Core. For information about the REST interface, refer to REST APIs and WS EMS on page 58.
- Lua Script Resource — An application for host devices that uses the Lua scripting language to integrate with them. The Lua Script Resource is an optional component. You do not need it to run the WS EMS. As an alternative, you can write your own application that uses HTTP and communicates directly with the WS EMS. For information about the Lua Script Resource, refer to Working with the Lua Script Resource on page 77.

WS EMS and ThingWorx Core

The WS EMS is a stand-alone application that is installed on a remote device, and establishes AlwaysOn™, bidirectional communications between the device and ThingWorx Core. The WS EMS uses a small footprint, and supports a variety of operating systems and architectures. This flexibility allows the WS EMS to work with a large number of devices to provide an easy way to establish communication between an edge device and ThingWorx Core.

The Connection Sequence

The process of connecting to ThingWorx Core consists of three main steps: Connect, Authenticate, and Bind. The WS EMS performs the first two steps. Then, the WS EMS works with the Lua Script Resource or a custom application to perform the third step. Here are the steps in more detail:

1. Connect — The WS EMS opens a physical WebSocket to ThingWorx Core, using the host and port specified in its configuration file. If configured, TLS is negotiated at this time.
2. Authenticate — The WS EMS sends an authentication message to ThingWorx Core. This message must contain an Application Key that was previously generated by ThingWorx Core.

   Upon successful authentication, the WS EMS can interact with ThingWorx Core according to the permissions applied to its Application Key. For the WS EMS, this implies that any client that makes HTTP calls to its REST interface can access functionality on ThingWorx Core. For this reason, the WS EMS is set by default to listen for HTTP connections on `localhost` (port 8000). You can change this listening port in the configuration file for the WS EMS.

At this point, WS EMS can make requests to ThingWorx Core and interact with it, much like an HTTP client can interact with the REST interface of ThingWorx Core, but it cannot direct requests to the Edge device.

3. Bind — To enable ThingWorx Core to send requests to it, the WS EMS works with the Lua Script Resource (or custom application) to send a BIND message to ThingWorx Core on behalf of the devices. Note that this step is optional, if you do not want the devices to receive and process requests from ThingWorx Core. It is required if you want to transfer files from ThingWorx Core to your devices or if you want to use tunneling.

   The BIND message can contain one or more names or identifiers for the devices. Note that corresponding *remote things* must have been created on ThingWorx Core to represent your edge devices. Remote things are things that are created using the **RemoteThing** thing template (or one of its derivatives) in ThingWorx Composer. When it receives the BIND message, ThingWorx Core associates the matching remote things in its database with the WebSocket that received the BIND message. This association allows the ThingWorx Core to use the WebSocket to send requests to the edge devices, and update the **isConnected** and **lastConnection** time properties for the corresponding remote things on ThingWorx Core.

   The WS EMS can send an UNBIND message to ThingWorx Core that removes the association between the remote things stored there and the WebSocket. The **isConnected** property is then updated to **false**.

Deployment

Once you have properly configured and integrated the ThingWorx WS EMS and Lua Script Resource (or your custom application), you can deploy them in one of the following ways:

- Embedded Deployment — Integrate the WS EMS and Lua Script Resource (or your custom application) directly into the application software stack of the edge device.
- Tethered Deployment — Deploy the WS EMS to a simple black-box that

connects to the diagnostic and sensor ports of an intelligent device. Deploy the Lua Script Resource or your custom application either on the same black-box, or on the intelligent device.

- Networked Gateway Deployment — Deploy the WS EMS on a simple server appliance that exists on the same network as a set of intelligent devices (for example, sensor networks or clusters of network-capable equipment),. Then, deploy the Lua Script Resource or your custom application on other hardware on the same local network.

Configuration Overview

To start connecting your edge devices to ThingWorx Core, you need to do the following:

1. Begin the initial configuration of the ThingWorx WS EMS, as described in Getting Started with the ThingWorx WS EMS on page 16
2. Once you have successfully connected to ThingWorx Core, complete the full configuration of the WS EMS according to your needs. Refer to the section, Additional Configuration of the ThingWorx WS EMS on page 28.
3. To use the Lua Configuration Script to host remote things for integration with ThingWorx Core, begin the initial configuration of the Lua Script Resource, as described in Getting Started with the Lua Script Resource on page 78.

Features

The ThingWorx WS EMS includes the features that are described in the sections below. Together, these features allow your edge devices to communicate with ThingWorx Core.

AlwaysOn™ Protocol

The ThingWorx AlwaysOn™ protocol is a binary protocol that uses the WebSocket protocol as its transport. The WS EMS uses the AlwaysOn protocol for communications with ThingWorx Core. This protocol provides a number of benefits:

- The devices that are running a WS EMS initiate all connections, which eliminates the need to open ports for inbound connections if the edge devices are deployed behind a firewall.
- The AlwaysOn protocol uses HTTP and the standard HTTP/HTTPS ports (80 and 443) to initiate and maintain connectivity, which eliminates the need for opening secondary ports for outbound communications.
- The protocol supports the TLS standard for securing the connection to ThingWorx Core.

- Once a connection is established, AlwaysOn binary messages are passed between the edge device and ThingWorx Core. AlwaysOn binary messages do not require re-initiating the HTTP connection for each request and therefore do not require the additional overhead of the standard HTTP messages. A ping/pong exchange of messages between a WS EMS and ThingWorx Core keep the connection alive during periods when the connection might be closed due to inactivity..
- The connections are persistent, which allows ThingWorx Core to make outbound requests to an edge device. ThingWorx Core can send requests to read or write properties, and to invoke services at the device, all with very low latency.

Security

The following default settings for the configuration of WS EMS support secure communications:

- Encryption — By default, the WS EMS always attempts to connect to ThingWorx Core using TLS.
- Certificates — By default, the WS EMS attempts to validate the certificate presented by ThingWorx Core during TLS negotiation.

HTTP Interface for REST APIs

In addition to the AlwaysOn interface, the WS EMS has an HTTP interface that supports REST API calls. This HTTP interface allows other applications to interact with ThingWorx Core through the AlwaysOn connection of the WS EMS. Since this other interface is HTTP, a custom application or the Lua Script Resource can be on a machine that is separate from the WS EMS and still communicate with it. The HTTP/REST interface of the WS EMS is a reflection of the REST interface of ThingWorx Core.

Lua Script Resource

The optional Lua Script Resource is a statically linked application that is used to run Lua scripts and configure things (devices) for integration with the host system.

2

Working with the ThingWorx WS EMS

Getting Started with the ThingWorx WS EMS

This section provides an overview of how to install, initially configure, and run the ThingWorx WS EMS.

Installing the ThingWorx WS EMS Distribution

The ThingWorx WS EMS is available from PTC and is distributed as a `.zip` file.

To install the package, follow these steps:

1. Download the package that is correct for the operating system and hardware platform that you plan to use:
   - MicroServer-*version*-Linux-ARM-HWFPU
   - MicroServer-*version*-Linux-ARM
   - MicroServer-version-Linux-coldfire
   - MicroServer-*version*-Linux-x86-32
   - MicroServer-*version*-Linux-x86-64
   - MicroServer-*version*-Win32
2. After downloading the package, select a location for the WS EMS distribution archive on the file system of the edge device.
3. Unzip the distribution archive.

WS EMS Distribution Contents

When unzipped, the ThingWorx WS EMS distribution creates the folder, `\MicroServer`, which contains the following files and subdirectories:

Item	Description
`wsems`	The WS EMS executable that is used to run the Edge MicroServer. **Note** Linux users must be granted permissions to this file. If you require FIPS support on the supported Windows platforms (win32), make sure you have the FIPS package, which contains the `ws-ems-fips.exe` file.
`luaScriptResource`	The Lua utility that is used to run Lua scripts, configure remote things, and

Item	Description
	integrate with the host system.. **Note** Linux users must be granted permissions to this file. If you require FIPS support on supported Windows platforms (win32), make sure you have the FIPS package, which contains the file,`luaScriptResource-fips.exe`, instead.
`\etc`	Directory that contains configuration files and directories for the luaScriptResource utility.
`\etc config.json.complete`	A reference file that contains all the configuration options available for the WS EMS. **Note** The `config.json.complete` file is intended to be used only as an example of the structure of a configuration file. It cannot be used as presented in your environment.
`\etc config.json.minimal`	A reference file that contains the basic settings that are required to establish a connection.
`\etc config.lua.example`	A reference file that contains a basic configuration for the luaScriptResource utility. A `config.lua` file is required to run the Lua engine.
`\etc\community`	Directory from which third-party Lua libraries are deployed. Examples of these libraries include the Lua socket library and the Lua XML parser, .
`\etc\custom`	Directory that will contain your custom scripts and templates.
`\etc\custom\templates`	Directory that contains an example `.lua` template and that is used to deploy custom templates.
`\etc\custom\scripts`	Directory from which custom integration scripts are deployed.
`\etc\thingworx`	Directory that contains ThingWorx-specific Lua files that are used by the Lua Script Resource (LSR). Do not modify this directory and its contents because an upgrade will overwrite any changes.
`\install_services`	Directory that contains the `install.bat` file for Windows bundles or the following files for Linux bundles: `install`, `tw_luaScriptResourced`, and `tw_microserverd`.

Libraries

The ThingWorx WS EMS uses the following libraries on Linux platforms:

- libpthread
- libstdc++
- libgcc_s
- libc
- libm (math library)

The Lua Script Resource also has libdl (dynamic loader).

Versions of the Libraries for Supported Platforms

The following table shows the supported platforms for WS EMS and the versions of the libraries that you can use with them:

Platform	libc	libpthread	libstdc++	libgcc	libm	libdl (LSR only)
Linux ARM	2.9 (with gcc version 4.3.3)	2.9	6.0.10	4.3.3	2.9	2.9
	2.8 (with gcc version 4.6.0)	2.9	6.0.10	4.3.3	2.9	2.8
Linux ARM HWFPU	2.8	2.8	6.0.15	4.6.0	2.8	2.8
Linux x86–32	2.8	2.8	6.0.10	4.3.2	2.8	2.8
Linux x86–64	2.8	2.8	6.0.15	4.6.0	2.8	2.8
Linux coldfire	2.11.1	2.11.1	6.0.14	4.5.1	2.11.1	2.11.1
Win32	XP or later	n/a	n/1	n/a	n/a	n/a

Running the ThingWorx WS EMS

The ThingWorx WS EMS can be run either from a command line or as a service to establish a connection to ThingWorx Core.

Running from a Command Line

The ThingWorx WS EMS can be run from a command line as follows:

1. Open a command window or terminal session on the system or device that is hosting the WS EMS.
2. Change to the directory, `\MicroServer\etc`.
3. For a basic configuration, copy the file, `config.json.minimal`, and rename it to`config.json`. For a complex configuration, copy the file, `config.json.complete` and rename it to `config.json`.

Note

The `config.json.complete` file is intended to be used only as an example. It cannot be used to run the WS EMS in your environment.

4. Set the configuration parameters:
    - For a simple configuration that establishes a connection to ThingWorx Core, see the section, Connecting to ThingWorx Core on page 21.
    - For a more complex configuration, start with the section, Connecting to ThingWorx Core on page 21, and then continue to the section, Additional Configuration of the ThingWorx WS EMS on page 28.
5. Change directories back to the top-level directory, `\MicroServer`.
6. To run the WS EMS, enter the command, `wsems`.
   The WS EMS attempts to connect to ThingWorx Core, and returns a message that the connection was successful to the console. You can tell that WS EMS is running and connected to ThingWorx Core by looking at the console prompt —two plus signs (++) indicate that it is running and connected.
7. Should you need to shut down the WS EMS, press ENTER to display the console prompt and type `q`.

Running as a Service

This topic explains how to run the ThingWorx WS EMS and ThingWorx Lua Script Resource as Windows services or Linux daemons.

To install WS EMS and LSR as services on a supported Windows or Linux platform:

1. Open a command window or terminal session on the system or device that is hosting the WS EMS.
2. Change to the `\MicroServer\etc` directory.
3. Set up the configuration file as necessary:
    - For a simple configuration to establish a connection to ThingWorx Core, see Connecting to ThingWorx Core on page 21.
    - For a more complex configuration, start with the simple configuration and then continue to the section, Additional Configuration of the ThingWorx WS EMS on page 28.

Note

The `config.json.minimal` and the `config.json.complete` files are intended to be used only as examples of the structure of a configuration file. Do not use them as presented in your environment.

4. Save the configuration file as `config.json`.

Note

The configuration file must be named `config.json` and reside in the `\MicroServer\etc` folder.

5. Follow the steps for your operating system:
   - For Windows:
     a. Change into the `\MicroServer\install_services\` directory.
     b. Run the following command to install WS EMS and LSR as Windows services, with or without the options to create custom names:

        ```
        install.bat [-wsems your_name_for_wsems_service_here]
        [-lsr your_name_for_lsr_service_here]
        ```

        Use only one hyphen (-) for the options. There is a space between the option and the argument, but for Windows, you must use the full option name.
   - For Linux:
     a. Change to the `\MicroServer\install_services\` directory.
     b. To make the install script executable, use the following command:

        ```
        sudo chmod +x MicroServer/install_services/install
        ```

     c. Run one of the following commands to install the WS EMS and LSR as daemons:

        Linux, using short names for the options

        ```
        sudo ./MicroServer/install_services/install \
              [-w your_name_for_ws_ems_service_here] \
              [-l your_name_for_lsr_service_here]
        ```

        Use only one hyphen (-) for the options. There is a space between the option and the argument.

        Linux, using long names for the options:

        ```
        sudo ./MicroServer/install_services/install \
              [--wsems=your_name_for_ws_ems_service_here] \
              [--lsr=your_name_for_lsr_service_here]
        ```

Note that the long options require two hyphens (`--`) before the option name, an equal sign following the name, and no space between the equal sign and the argument (your name for the service).

Note

To uninstall the service, remove the service from `/etc/init.d/<Service Name>`.

Connecting to ThingWorx Core

This section explains how to configure a WS EMS to connect your device to ThingWorx Core. Additional configuration options are explained later.

Creating a Configuration File

The configuration file of WS EMS is a text file that uses the JSON format. It is separated into multiple sections. Each section contains sets of name/value pairs (parameters) that control different aspects of the configuration. To connect to ThingWorx Core, only a few parameters are required.

To view an example of a basic configuration file and create your own configuration file:

1. From the WS EMS installation directory, change to the directory, `/etc`.
2. Open `config.json.minimal`in a text editor. This file contains the minimum set of configuration settings required to connect to ThingWorx Core
3. Create a new file in your editor, and save it as `config.json`.

Connecting

To connect to ThingWorx Core, you must configure the `ws_servers` section in the configuration file. This section contains the parameters that define the connection between the ThingWorx WS EMS and ThingWorx Core. You need to provide an IP address or host name and port for one ThingWorx Core instance.

Note

Previous releases of the WS EMS allowed you to configure an array that contained multiple addresses. However, the WS EMS no longer checks for another address if it fails to connect with the first address in the array. If you previously specified multiple addresses, you do NOT have to change your configuration file. The WS EMS will use the first address in the `ws_servers` array and ignore the rest.

As long as you have created your own `config.json` file, follow these steps to set up the connection:

1. Copy the first two lines from the `config.json.minimal` file and paste them in your file:

   ```
   {
     "ws_servers" :  [
   ```

   You are ready to add the parameters that define the connection.
2. Under `"ws_servers"`, add the `"host"`and `"port"`parameters. Then, type the URL of the host computer and the port on that computer to use for the connection. If the connection is to be secure, use port 443. For example:

   ```
   {
    "ws_servers": [
         {"host": "acmeThingWorxHost", "port": 443 }
    ],
    "appKey": "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx"
   }
   ```

   For development purposes, you may want to use a ThingWorx Core instance that is on the same computer where you installed your WS EMS. `"localhost"`can be used as the value for `"host"`for these purposes only.
3. Add the `appKey`parameter (shown above), and copy/paste the Application Key that was generated for the WS EMS to use to access ThingWorx Core. A ThingWorx administrator can generate Application Keys. The key is associated with an account and determines the privileges that the WS EMS will have when accessing ThingWorx Core.

Enabling Encryption

You enable or disable encryption in the `ws_encryption` section of the configuration file. By default, the ThingWorx WS EMS always attempts to connect to ThingWorx Core using TLS (that is, encryption is enabled).

Note

The code samples below are provided for example purposes only.

To enable encryption, specify the parameters as shown below:

```
"ws_connection": {
  "encryption": "ssl"
}
```

To disable encryption, specify the parameters as shown below:

```
"ws_connection": {
  "encryption": "none"
}
```

Parameters

The `ws_encryption` section contains the following parameter:

Use	To Specify
`encryption`	Whether or not encryption is enabled for communications with ThingWorx Core, and the type of encryption used. Valid values are: • `none` • `ssl` • `fips` **Note** The `fips` value is available only if you installed the FIPS version of the WS EMS.

Configuring SSL/TLS Certificates and Validation

Use the `certificates` section of the configuration file to specify whether certificates will be validated for the connection between the ThingWorx WS EMS and ThingWorx Core.

The WS EMS can validate certificates that have been signed using the following algorithms:

- MD5
- SHA-1
- SHA-256 digest

Note

Two versions of the WS EMS distribution are available. One has a built-in AxTLS library for secure connections. The other provides the OpenSSL toolkit and FIPS module and has `-fips` in the name of the distribution file.

It is important to note that the AxTLS library does not support client authentication when it is used in a client-side application (i.e., for an edge device). It does support it when used in a server-side application. If you require client authentication for the WS EMS, you must use the WS EMS distribution that contains the OpenSSL libraries instead of the one with the AxTLS library. The currently supported version of OpenSSL is 1.0.1h. The use of OpenSSL with the WS EMS works on Windows platforms only at this time. It does not work on Linux platforms.

The code examples below are for example purposes only.

By default, the WS EMS attempts to validate certificates presented by ThingWorx Core during TLS negotiation. If necessary, you can disable this validation by setting the `validate` parameter to `false`, as shown below:

```
"certificates": {
  "validate": false
}
```

For testing purposes, you may want to use self-signed certificates on your server. However do NOT use them in a production environment. By default use of self-signed certificates is disabled. To allow self-signed certificates, use the following setting:

```
"certificates": {
  "allow_self_signed": true
}
```

To use a certificate that has been signed by a Certificate Authority, you must provide that Authority’s root certificate in PEM or DER format and indicate the path to the certificate, as shown below:

```
"certificates": {
  "validate": true,
  "allow_self_signed": false,
  "cert_chain": ["/path/to/ca_cert/file"]
}
```

The following table lists and describes the parameters for configuring SSL/TLS certificates and validation:

Use	To Specify
`validate`	Whether or not to perform validation on certificates presented by ThingWorx Core. For production, enabling validation is strongly recommended.
`allow_self_signed`	Whether or not to permit self-signed certificates to be used. For production, allowing self-signed certificates is not recommended.
`cert_chain`	The path to the certificate file. **Note** Multiple certificates can be loaded into this array.
`client_key (optional)`	The RSA key used for client-side certificates. The key can be encrypted.
`client_cert (optional)`	The path to the X.509 certificate file for the client.
`key_file (optional)`	The path to the key file to load for client certificates (supports .pem, .p8, and .p12 files).
`key_passphrase (optional)`	A string password for opening the key file. The passphrase can be encrypted.

Validation Criteria

To validate a certificate, you can configure the fields of the certificate that should be validated:

- Subject common name
- Subject organization name
- Subject organizational unit
- Issuer common name
- Issuer organization name
- Issuer organizational unit

When creating a Certificate Signing Request, you are prompted for the fields that will need to be validated. The following definitions may help you determine the values for these fields:

- Common Name — Typically, a combination of the host and domain names, the value of these fields looks like “www.your_site.com” or “your_site.com”. Certificates are specific to the Common Name that they have been issued to at the Host level. This Common Name must be the same as the web address that the WS EMS will access when connecting to ThingWorx Core using TLS. If ThingWorx Core and the WS EMS are located on an intranet, the Common Name can be just the name of the host machine of ThingWorx Core.

- Organization Name — Typically, the name of the company.
- Organizational Unit — Typically, a department or other such unit within a company. For example, IT.

The code example below shows how to specify values for these fields.

FIPS Mode

To use FIPS mode, you need to download the WS EMS distribution that has `-fips` in the name of the file. This distribution includes the OpenSSL toolkit, which the WS EMS uses in conjunction with the OpenSSL FIPS Object Module 2.0.2 (Windows only). With this distribution, FIPS is automatically enabled so that your application can use an embedded FIPS-140-2-validated cryptographic module (Certificate #1747; OpenSSL FIPS module version 2.0.2) on all supported platforms per FIPS 140-2 Implementation Guidance section G.5 guidelines.

Note

Not all hardware platforms where applications written using the WS EMS can support FIPS-140-2-validated cryptography. For example, on platforms based on IA32 architecture, the processor must support the SSE2 instruction set. The SSE2 instruction set is available in Intel x86 CPUs, starting with Pentium 4. The application log will have a message that FIPS-140-2-validated cryptography is enabled. When you choose the FIPS distribution of WS EMS, be sure that your certificates include only FIPS approved encryption algorithms. The FIPS approved algorithms are AES, Triple-DES, RSA, DSA, DH, SHA1, and SHA2.

If you use the FIPS distribution and your application directly communicates with a Java-based SSL/TLS server (such as ThingWorx Core), the cipher suite list should include `!kEDH` (as shown below). Otherwise, ephemeral Diffie-Hellman (EDH) key exchange may fail:

```
<CipherSuites>DEFAULT:!kEDH</CipherSuites>
```

Example

Examples of the `certificates` and `validation_criteria` sections are shown below:

```
 "certificates": {
 "validate": true,
"allow_self_signed": false,
 "cert_chain": ["/path/to/ca_cert/file"]
 "client_key": "AES:EncryptedKey",
 "client_cert": "/path/to/client/cert/file",
 "key_file":  "/path/to/key/file",
 "key_passphrase": "AES:EncryptedPassphrase",
}
"validation_criteria": { // Object specifying which fields in the server
                         // certificate to validate, and the expected values
    "Cert_Common_Name": "common name", // subject common name
    "Cert_Organization": "organization", // subject organization name
    "Cert_Organizational_Name": "organizational unit", // subject organizational unit
    "CA_Cert_Common_Name": "cert common name", // Issuer common name
    "CA_Cert_Organization": "cert organization", // Issuer organization name
    "CA_Cert_Organizational_Name": "cert organizational unit" // Issuer organizational unit
  }
```

Authenticating and Binding

The `appKey`section of the configuration file is used for authentication. The ThingWorx WS EMS must be authenticated to connect to ThingWorx Core.

An Application Key is an authentication token generated by ThingWorx Core that represents a specific user. The Application Key is sent along with the connection request to authenticate the WS EMS with ThingWorx Core, and apply the correct permissions that are associated with the Application Key's user account.

The Application Key is set by a simple top-level key in the JSON structure.

Note

The code sample below is provided for example purposes only.

```
"appKey": "37b62268-592d-486a-9493-d62a963e3515"
```

Binding

Once another device registers with it (by sending an auto bind message), WS EMS sends a BIND message to ThingWorx Core on behalf of that device. ThingWorx Core then associates the WebSocket on which the WS EMS is communicating with a remote thing on ThingWorx Core whose name or identifier matches the name or identifier sent by the WS EMS (the remote thing must have been created on ThingWorx Core, using the **RemoteThing** thing template or one of its derivative templates). This association is the binding that must exist so that ThingWorx Core can send requests to the device that is communicating through the WS EMS. For information about automatic binding and the WS EMS, refer to the section, Configuring Automatic Binding for WS EMS on page 36. For information about configuring the gateway option for automatic binding, refer to Auto-bound Gateways on page 37.

Verifying Your Connection

Following the initial message that indicates a successful connection to ThingWorx Core, you can verify your connection as follows:

1. Open a web browser on the system or device that is hosting the WS EMS, and enter the following URL in the address bar:

   ```
   http://localhost:8000/Thingworx/Things/LocalEms/Properties/isConnected
   ```

   This request returns an infotable that contains a single row that indicates whether the WS EMS is online or offline.
2. If the result of this request is true, the WS EMS is online. To test that you can access data on ThingWorx Core, enter the following URL in the address bar:

   ```
   http://localhost:8000/Thingworx/Things/SystemRepository/Properties/name
   ```

   This request returns an infotable that is serialized to JSON, where the `row` of the infotable contains the name of the requested thing, “SystemRepository”.
3. If both of these tests succeed, the WS EMS is successfully connected to ThingWorx Core.

Additional Configuration of the ThingWorx WS EMS

This section describes how to configure additional options of the ThingWorx WS EMS.

The `config.json.complete` file provided in the WS EMS installation contains all possible configuration parameters. Do not attempt to use it as is. It is intended as a reference.

Some parameters in the `config.json.complete` file have default values that you may not need to change. This section describes the parameters that are used most often.

Note

If you are using the Lua Script Resource (`luaScriptResource.exe`) to communicate with one or more local devices, you also need a `config.lua` in the `/etc`directory. This file tells the Lua process how to initialize and communicate with ThingWorx Core through the WS EMS. For more information, see Lua Script Resource Configuration File on page 80.

Viewing All Options

The file, `config.json.complete`, is provided in the WS EMS distribution. It contains all the possible options that you can configure for your WS EMS. To view these options, follow these steps:

1. From the WS EMS installation directory, change to the directory, `/etc`.
2. Open the file, `config.json.complete` in a text editor.
3. Refer to this file while you read about the sections of the configuration file.

The comments in this file provide brief descriptions of the parameters. The rest of this section walks you through the parameters that are used most often.

Configuring the Logger Section

Use the `logger` settings to configure a WS EMS to collect logging information. Here is an example of this section:

```
{"logger":
    "level":"AUDIT",
    "audit_target":"http://test.com",
    "publish_directory":"/_tw_logs/",
    "publish_level":"ERROR",
    "max_file_storage":2000000,
    "auto_flush":true
},
```

The following table lists and describes the parameters of the `logger` element :

Use	To Specify
`level`	The level of information that you want to include in the audit log file. Valid values include: • `AUDIT` • `ERROR` • `WARN` • `INFO` • `DEBUG` • `TRACE` **Tip** When troubleshooting a problem, set the `level` to `TRACE` so that you can see all the activity. For production, set the `level` to `ERROR` if you want to see error messages.
`audit_target`	The path to the audit log file where audit events will be written. Alternatively, specify an HTTP address for the audit log file, where these events will be sent using a POST command.

Use	To Specify
	Audit events are also written to the normal log destination . If no target is specified, no additional auditing takes place. Valid values include: • file://*path_to_file* • http://*hosted_location*
`publish_directory`	A location for writing alternate log files for logging information that meets or exceeds the `publish_level`. If you do not specify a location for this parameter, this logging information is not written to alternate log files.
`publish_level`	The level of information that you want to include in the alternate log files. Valid values include: • `AUDIT` • `ERROR` • `WARN` • `INFO` • `DEBUG` • `TRACE`
`max_file_storage`	The maximum size of the log file before the log begins writing to the alternate log file. Keep in mind that there are two concurrent log files. The total disk space taken when both files reach the maximum size is the value of `max_file_storage` times 2.
`auto_flush`	Whether WS EMS should flush with every write to the `publish_directory` A setting of `true` forces WS EMS to flush with every write. **Note** Flushing with every write has a negative impact on performance.

Configuring the HTTP Server Section

The `http_server` section is used to allow a WS EMS to accept local calls from the machine or device code that may be running in a different process, such as the Lua Script Resource.

Note

If you are connecting to the Lua Script Resource and have changed the `host` and `port` parameters below, you must update the `config.lua` file (in the `/MicroServer` folder) and modify the parameters, `scripts.rap_host` and `scripts.script_resource_host`.

Here is an example of the `http_server` section. Change the values of the parameters to fit your environment; do not use this example as is.

```
 "http_server":  {
     "host": "localhost",
     "port": 8000,
     "ssl": false,
     "certificate": "/path/to/certificate/file",
     "authenticate": false,
     "content_read_timeout": 20000
    }
```

The following table lists and describes the parameters available in the `http_server` section :

Use	To specify
`host`	The name of the host that the WS EMS listens to. The default value is `localhost`, but this likely means IPV6, and not 127.0.0.1.
`port`	The port number on which the WS EMS listens. The default port is 8000.
`ssl`	Whether or not to use TLS for communications between ThingWorx Core and WS EMS. The default values is false.
`certficate`	The complete path to the certificate file if certificates are used in communications between ThingWorx Core and the WS EMS.
`authenticate`	Whether or not to enable authentication between the HTTP server and the WS EMS. The default value is false.
`content_read_timeout`	The maximum amount of time (in milliseconds) that the WS EMS will wait before timing out when it tries to read a PUT or POST. The default value is 20000.

To configure an IP address other than `localhost` for REST API calls, refer to the section, Configuring WS EMS to Listen on IP Other Than localhost on page 65

Configuring the WebSocket Connection

The `ws_connection`section is used to configure the WebSocket connection between the WS EMS and ThingWorx Core. If the default settings meet your needs, there is no need to include this section in the configuration file.

Note

The code sample below is provided for example purposes only. It shows the section as it appears in the `config.json.complete` file.

```
 "ws_connection": {
"ssl "encryption":
     "binary_mode": true,
     "verbose": false,
     "msg_timeout":  5000,
     "ping_rate": 55000,
     "pingpong_timeout": 10000,
     "connect_period": 60000,
     "duty_cycle": 100,
     "message_idle_time": 50
     "max_msg_size": 1048576,
     "message_chunk_size": 8192,
     "max_frame_size": 8192,
     "max_messages": 500,
     "connect_on_demand": false,
     "connect_timeout": 10000,
     "connect_retry_interval": 10000,
     "max_threads": 4,
     "max_connect_delay" : 0
     "socket_read_timeout": 100,
     "frame_read_timeout": 10000,
     "ssl_read_timeout": 500,
```

The following table lists and describes parameters available to configure the WebSocket connection, in alphabetical order:

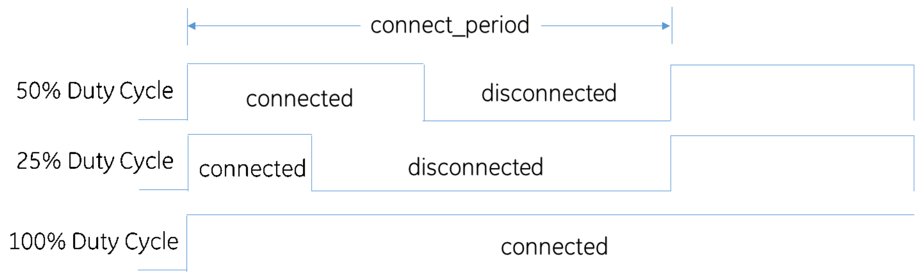
Use	To Specify
`binary_mode`	Whether or not all messages are sent using binary WebSockets. The default value is `true`.
`connect_on_demand`	Whether or not to connect automatically in order to send a message, even if in the off time of the duty cycle. The default value is `false`, which means do NOT connect automatically while in the off time of a duty cycle..
`connect_period`	The rate in milliseconds to duty cycle control the WebSocket connection. The default value is `60000` milliseconds.
`connect_retry_ interval`	The amount of time, in milliseconds, to wait between attempts to connect to ThingWorx Core. The default value is `10000`milliseconds (10 seconds).
`connect_timeout`	The maximum amount of time, in milliseconds, to wait for a connection to ThingWorx Core to be established. The default value is `10000` milliseconds (10 seconds).
`duty_cycle`	The percentage of time for the duty cycle of the WebSocket connection. The default value is `100`, for AlwaysOn. For more information, see the section, Duty Cycle Modulation on page 34, below.
`encryption`	Whether or not SSL is used for the WebSocket connection. The default value is `ssl`.
`frame_read_timeout`	The maximum amount of time, in milliseconds, to wait for a full SSL frame during a socket read operation. The default value is 10000 milliseconds. A timeout if the WS EMS requests something from ThingWorx Core and receives no data at all The timeout triggers an error and causes a

Use	To Specify
	disconnect.
`max_connect_delay`	The maximum amount of random delay time, in milliseconds, to wait before connecting. The default value is `0` milliseconds (no delay before connecting).
`max_frame_size`	The maximum size, in bytes, of a WebSocket frame. The default value is `8192` bytes.
`max_messages`	The maximum number of requests that can be waiting for a response at any one time. The default value is `500` requests.
`max_msg_size`	The maximum size, in bytes, of a complete message, even if broken into frames. The default value is `1048576` bytes (1 MB).
`max_threads`	The maximum number of incoming message handler threads to use. The default value is 4.
`message_chunk_size`	The maximum size of a chunk, of a large message that has been broken up into chunks. The default value is `8192` bytes.
`message_idle_time`	The time in milliseconds to wait to see if messages are being sent before disconnecting for the off time of the duty cycle. The default value is `50` milliseconds.
`msg_timeout`	The time in milliseconds to wait for a response to return from ThingWorx Core. The default value is `5000`milliseconds (5 seconds).
`ping_rate`	The rate in milliseconds to send pings to ThingWorx Core. The default value is `55000`milliseconds.
`pingpong_timeout`	The amount of time in milliseconds to wait for a pong response after sending a ping. A timeout initiates a reconnect. The default value is `10000` milliseconds.
`socket_read_timeout`	The maximum amount of time, in milliseconds, to wait for data before timing out. The default value is `100` milliseconds. A timeout does not trigger an error. This parameter defines how long a process is allowed to wait for data before it must try again. Another process would be allowed to read from the socket between tries.
`ssl_read_timeout`	The maximum amount of time, in milliseconds, to wait for a full SSL record during a socket read operation. The default value is `500` milliseconds. A timeout does not trigger an error. This parameter effectively allows a read to continue after the `socket_read_timeout` is reached IF a partial amount of the SSL record is received on the socket before the `socket_read_timeout` expires.
`verbose`	Whether or not the WS EMS is in extremely verbose logging mode. The default value is `false`.

Duty Cycle Modulation

Duty cycle modulation defines the frequency and duration of the connection between a WS EMS and a ThingWorx Core instance. If you need to conserve power or bandwidth at the expense of availability/responsiveness, you can use duty cycle modulation to place theWS EMS into a sleep mode by setting the following parameters:

- `duty_cycle`— Determines what percentage of time during the connection period that the WS EMS is connected to ThingWorx Core.
- `connect_period` — Defines the period of time set for duty cycle intervals.



Configuring a Proxy Server

The `proxy`section is used to configure the WS EMS to use a proxy server to connect to ThingWorx Core. For authentication with a proxy server, the WS EMS supports the following options:

- No authentication
- Basic authentication
- Digest authentication
- NTLM

Note

The code sample below is provided for example purposes only.

```
"proxy" : {
    "host" : "localhost",
    "port" : 3128,
    "user" " "username",
    "password": "AES:EncryptedPassword"
},
```

The following table lists and describes the parameters for setting up a proxy server:

Use	To Specify
`host`	The host name of the proxy server used to connect to ThingWorx Core. The value of the host parameter can be an IP address, or a host name.
`port`	The port number used to communicate with the proxy server. The default value is 3128.
`user`	The name of the user account that is used to connect with the proxy server.
`password`	The password of the user account that is used to connect with ThingWorx Core. If you select a user name and password combination, it is recommended that you encrypt the password as shown in the example above. To encrypt a password, you can use the `Encrypt` method of the **Encryption** function in the service editor of the server to return the encrypted string value.

Storing Messages Received While WS EMS Is Offline

The `offline_msg_store` section is used to configure how the WS EMS handles and stores messages that are received while it is offline.

The following example shows the default configuration. If you do not want to use it, you do not have to change the configuration. When this feature is disabled, the other settings are ignored. However, if you do want to use this store, make sure you set the `enabled` parameter to `true` and add the appropriate directory for storing messages. Depending on the space available on your system, you may also want to change the `max_size` parameter.

```
"offline_msg_store": {
     "enabled": false,
     "directory": "<startup_dir>/offlineStorage",
     "max_size": 65535
    }
```

The following table lists and describes the parameters for storing messages that are received while the WS EMSis offline; it also provides the default values:

Use	To Specify	Default Value
`enabled`	Whether or not the store is enabled.	`false` (This store is disabled.)
`directory`	The path to the directory of the store where messages are stored.	`<startup_dir>/offlineStorage`, where `<startup_dir>` is the directory where the WS EMS executable is installed.
`maxsize`	The maximum size (in bytes) of the directory where messages are stored.	`65535`

Configuring Automatic Binding for WS EMS

The `auto_bind` section is used to define specific local things that are always expected to be bound through this WS EMS, or to define a WS EMS as a gateway. For more information about configuring WS EMS as a gateway, refer to Auto-bound Gateways on page 37.

The `auto_bind` parameter is an array, allowing you to statically define more than one device or machine thing.

Note

The code sample below is provided for example purposes only.

```
"auto_bind": [{
"EdgeThing001","name":
"localhost","host":
8001,"port":
"/", "path":
     "timeout": 30000,
     "protocol": "http | https",
     "user": "username",
     "password": "AES:EncryptedPassword",
     "accept": "application/json",
     "gateway": true/false
   }],
```

The following table lists and describes the parameters for automatic binding:

Use	To Specify
`name`	REQUIRED. The thing name of the entity as it exists on ThingWorx Core. If an identifier is configured for the thing in ThingWorx Core, you must specify that identifier here. For more information, see the section, “Identifiers”, below.
`host`	The name of the host for the thing. The default value is `localhost`, but this likely means IPV6, and not 127.0.0.1.
`port`	The port number used by the thing for communications. The default value is `8001`.
`path`	The path to prepend to the path received in ThingWorx Core request.
`timeout`	The maximum amount of time to wait for a response from the target, in milliseconds. The default value is `30000` milliseconds (30 seconds).
`protocol`	Whether the protocol to use for communications is HTTP or HTTPS. The default value is `HTTP`.
`user`	The name of the user account to use for authentication when connecting.

Use	To Specify
`password`	The password for the user account specified for the `user` parameter. If you specify a user name and password, it is recommended that you encrypt the password, as shown in the example above. To encrypt a password, you can use the `Encrypt` method of the **Encryption** function in the service editor of the server to return the encrypted string value.
`gateway`	Whether this automatically bound thing is a gateway or non-gateway thing. To understand the differences between these two settings, refer to Auto-bound Gateways on page 37.

Identifiers

Identifiers provide a way to specify an alternate name for a given thing. An identifier can be set for a thing on the **General Information** tab of the thing in ThingWorx Composer. If a thing has an identifier set, the WS EMS must bind the thing using the identifier. A typical use case for an identifier is the serial number for a device, as opposed to an intuitive name.

You can use an identifier when dynamically registering a thing or when configuring the `auto_bind` section. To use an identifier, prepend an asterisk (*) to the identifier and specify it as the value for the `"name"` parameter, as follows:

```
{
   "name" : "*SN0012",
   "host" : "localhost",
   "port" : 9000,
   "path" : "/"
 }
```

Auto-bound Gateways

When you configure the `auto_bind` section of a WS EMS configuration, it is very important to note the difference between the settings, `"gateway":true` and `"gateway":false`. When used with a valid `"name"`parameter, either value results in the WS EMS attempting to bind the thing with ThingWorx Core. In addition, either value allows the WS EMS to respond to file transfer and tunnel services that are related to the automatically bound things. However, the similarities end here.

Gateway

An auto-bound gateway can be bound to ThingWorx Core ephemerally if there is no thing to bind with on ThingWorx Core. Ephemeral binding is a temporary association between ThingWorx Core and the WS EMS that lasts only until the WS EMS unbinds the gateway. In general, ephemeral things are created on ThingWorx Core when it has no remote things with a matching thing name in its database.

When the WS EMS is attempting to bind a gateway, a thing is automatically created on ThingWorx Core, using the **EMSGateway** template. ThingWorx Core binds the auto-bound gateway with this ephemeral thing. This thing is accessible only through the ThingWorx REST API. Once the WS EMS unbinds the gateway, the ephemeral thing is deleted

If you do not want the automatically bound gateway to be ephemeral, you can create a thing for it and choose the **EMSGateway** thing template in ThingWorx Composer. If you do not choose this template for the thing, ThingWorx Core does not bind the gateway with your thing.

When used both normally and ephemerally, the **EMSGateway** template provides some services that are specific to gateways. These services are not accessible to things that are created with the **RemoteThing** (and derivatives) thing templates. Refer to the **EMSGateway Class Documentation** for more details. For a list of REST APIs that provide some of the services of the **EMSGateway** class, refer to Using Services with a WS EMS on page 68

Note

The devices for which a WS EMS acts as a gateway can be set up to identify themselves to the WS EMS when they initialize and connect to it. Alternatively, when these devices are well known, you may want to define them explicitly in the WS EMS configuration. For examples of these two types of gateway configurations for WS EMS, refer to the section, Example Configurations on page 47.

Non-Gateway

A thing that is automatically bound but is not a gateway has the following requirements:

- For the WS EMS to respond to messages that are related to properties, services, or events for a non-gateway, automatically bound thing, a LuaScriptResource must exist. Custom Lua scripts must exist within the LSR to provide the capabilities to handle services, properties, and events.
- For the non-gateway thing to bind successfully, you must first create a corresponding thing on ThingWorx Core, using the **RemoteThing** thing template (or any template derived from **RemoteThing**, such as **RemoteThingWithFileTransfer**).

The most common use of this type of automatically bound thing is to bind a simple thing that can handle file transfer and tunnel services but does not need any custom services, properties, or events.

Note

For an example of a non-gateway configuration for WS EMS, refer to the section, Example Configurations on page 47.

Configuring Settings for Tunneling

Application tunnels allow for secure, firewall-transparent tunneling of TCP client/server applications, such as VNC and SSH. As long as the WebSocket connection between the edge device and ThingWorx Core is secure (for example, uses a TLS certificate), the client/server applications can run securely. How is this possible? The application opens a second WebSocket to the same host and port that is used for other communications between the edge device and ThingWorx Core. You do not need to open other ports in the firewall to run these applications. However, it is important to note that it connects to a different URL that is specifically for the tunnel.

Note

Only TCP client/server applications are supported at this time. UDP is not supported.

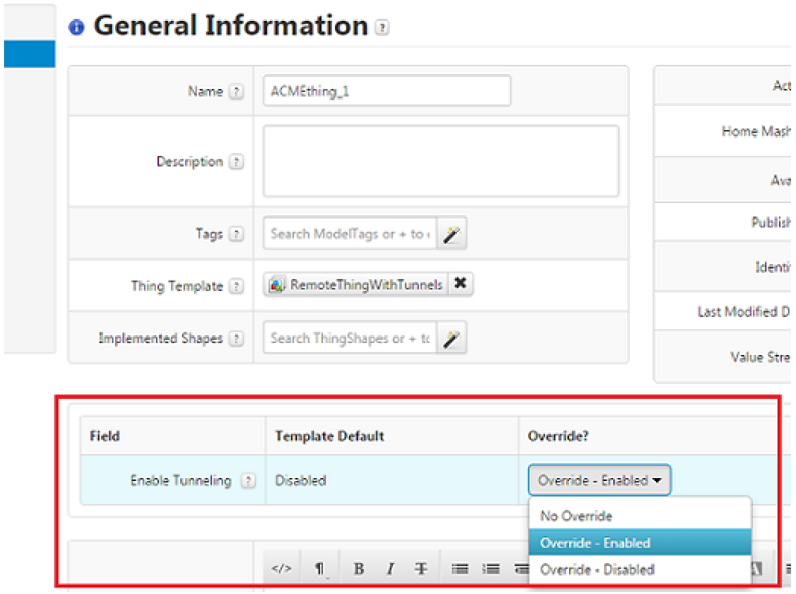
Configure tunneling for your WS EMS when you want to be able to access the edge device that is running WS EMS through a remote desktop session (for example, UltraVNC) or remote terminal session (for example, SSH) session. By default, tunneling is enabled for the WS EMS. If you are using UltraVNC or SSH, the default settings in the configuration file will suffice. The rest of the configuration takes place in the server side of the client/server application (UltraVNC Server, for example) and on ThingWorx Core, through ThingWorx Composer.

The main steps for the built-in client/server application (UltraVNC) follow:

1. If you have not already, install UltraVNC Server on the edge device where the WS EMS is running.
2. Access the **Admin Properties** configuration screen for UltraVNC Server and make sure that the following configuration parameters are set:
   - **Allow loopback connections** — Make sure that this check box is selected if you want to test the connection on the edge device itself (the VNC Viewer is installed on the same machine as the VNC Server).
   - **VNC password** — Type the password that VNC Viewer users must type to access this edge device remotely.
   - **Multi viewer connection** — Select the option, **Keep existing connections**, so that a new session with this edge device does not disconnect any existing VNC Viewer sessions.

The main steps in ThingWorx Composer follow:

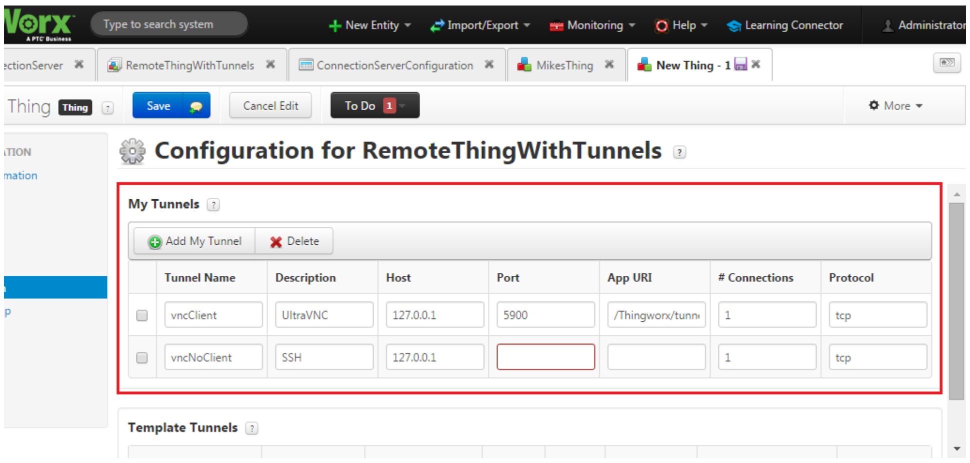
1. In the **Configuration** page for the Tunneling Subsystem, check the field, **Public host name used for tunnel**. If the IP address is a local network address, the tunnel will not work. Set this field to the external host/IP address that tunnels should use for connections. For more detail, see Required Setting for the Tunneling Susbsystem on page 44.
2. If you have not already, use the **RemoteThingWithTunnels** or the **RemoteThingWithFileTransferAndTunnels** template to create a remote thing on ThingWorx Core to represent the edge device that is running WS EMS.
3. After creating the thing, from the **General Information** page for the new thing, enable the template **Override?** setting for **Enable Tunneling**, as shown below. By default, this setting is disabled.


General Information

Name	ACMEthing_1
Description	
Tags	Search ModelTags or + to
Thing Template	RemoteThingWithTunnels
Implemented Shapes	Search ThingShapes or + tc

Field	Template Default	Override?
Enable Tunneling	Disabled	Override - Enabled

4. Determine which remote applications will be used to access the edge device and whether you want to use the VNC client that is built into ThingWorx Core. These applications may be any of the following types:
   - Desktop remote sessions — VNC Server on edge devices and corresponding Viewer client on the user machines that will access the device. The VNC Viewer is the built-in application available from ThingWorx Core. You could create a tunnel, using the name `vncClient`.
   - SSH — An SSH client/server application, such as PuTTY. For information on OpenSSH, refer to http://www.openssh.com/ or http://support.suso.com/supki/SSH_Tutorial_for_Linux. For information on PuTTY, visit http://www.putty.org/.
   - Microsoft RDP — Refer to the Microsoft web site, more specifically, http://windows.microsoft.com/en-us/windows/connect-using-remote-desktop-connection#connect-using-remote-desktop-connection=windows-7.
   - Custom client application that you have built
5. As long as you have enabled the template **Override?** setting for **Enable Tunneling** in the **General** tab for the remote thing, configure the tunnels for the thing that you created:
   a. Under **ENTITY INFORMATION**, select **Configuration**. If you are not in Edit mode for the thing, click **Edit**.
   b. Under **Configuration for RemoteThingWithTunnels**, click **Add My Tunnel** and in the displayed fields, enter the information for the client/server application. Here are examples for VNC and SSH:



Configuration for RemoteThingWithTunnels

My Tunnels

Add My Tunnel | Delete

	Tunnel Name	Description	Host	Port	App URI	# Connections	Protocol
☐	vncClient	UltraVNC	127.0.0.1	5900	/Thingworx/tunn	1	tcp
☐	vncNoClient	SSH	127.0.0.1			1	tcp

Template Tunnels

Tip

When configuring a mashup for the edge device, you will need to provide the names that you assign to the tunnels. To access the list of tunnel names available on a thing, use the **GetTunnelNames** service.

Configure the **Host** and **Port** fields from the point of view of the *edge device* where the server component of the client/server application is running, not ThingWorx Core. For example, when a user wants to access the edge device from VNC Viewer, the user would type the IP address of the device and then the port number 5900. For SSH, you might enter 22 in the field.

By default, the URL is the location of the VNC client application on ThingWorx Core. If you are using SSH, make sure that supply the port number and then make this field empty.

The values that you see for the **# Connections** and **Protocol** fields are the default values and are the only values that are currently supported.

6. Save the configuration of your new thing.
7. When creating your mashup, add a **Web Socket Tunnel** widget if you are using the built-in VNC viewer (client) that is provided with ThingWorx Core. At minimum, you need to set the following parameters for the **Web Socket Tunnel** widget:
   - **RemoteThingName** — You must supply the name of your thing (**RemoteThingWithTunnels**) that will use this tunnel (for example, the one that was created in Step 2 or earlier).
   - **TunnelName** — Enter the name that you assigned to the tunnel for the built-in VNC Viewer in the configuration of the thing.
   - **VNCPassword** — Type the password that the VNC Server that is running on the edge device will expect from VNC Viewer.

Note

If you do not see this widget, you need to download and import the `WebSocketTunnel_ExtensionPackage.zip`package into ThingWorx Core (see ThingWorx Core Help Center for your release and search for the widget by name).

If you are NOT using the built-in VNC Viewer, add a **RemoteAccess** widget. For example, you might use another type of TCP client/server application. For the **RemoteAccess** widget, set the following parameters:

- **RemoteThingName** — You must supply the name of your thing (**RemoteThingWithTunnels**) that will use this tunnel (for example, the one that was created in Step 2 or earlier).
- **TunnelName** — Enter the name that you assigned to the tunnel for the other type of application in the configuration of the thing (for example, PuTTY or another SSH client/server application).
- **ListenPort** — Enter the number of the port that the Java Web Start application will listen on when it starts up. For example, if you want to run an SSH session and the listen port is 9005, you would connect your SSH client to `localhost:9005`.
- **AcceptSelfSignedCerts** — If SSL/TLS is used for this connection and you are testing with a self-signed certificate, select the check box.

For complete information about configuring the **RemoteAccess** widget, refer to ThingWorx Core Help Center for your release of ThingWorx Core and search for “RemoteAccess widget”.

8. Save your mashup.
9. For WS EMS, tunneling is enabled by default. As long as your WS EMS is running and connected to ThingWorx Core, you can test your mashup.

The `tunnel` section of the WS EMS configuration file can be used to configure the buffer size, timeouts, and the number of concurrent sessions for tunneling. These settings suffice for most situations and may be overwritten by ThingWorx Core when a tunnel is initialized.

Note

The code sample below is provided for example purposes only.

```
"tunnel": {
     "buffer_size": 8192,
     "read_timeout": 10,
     "idle_timeout": 300000,
     "max_concurrent": 4
```

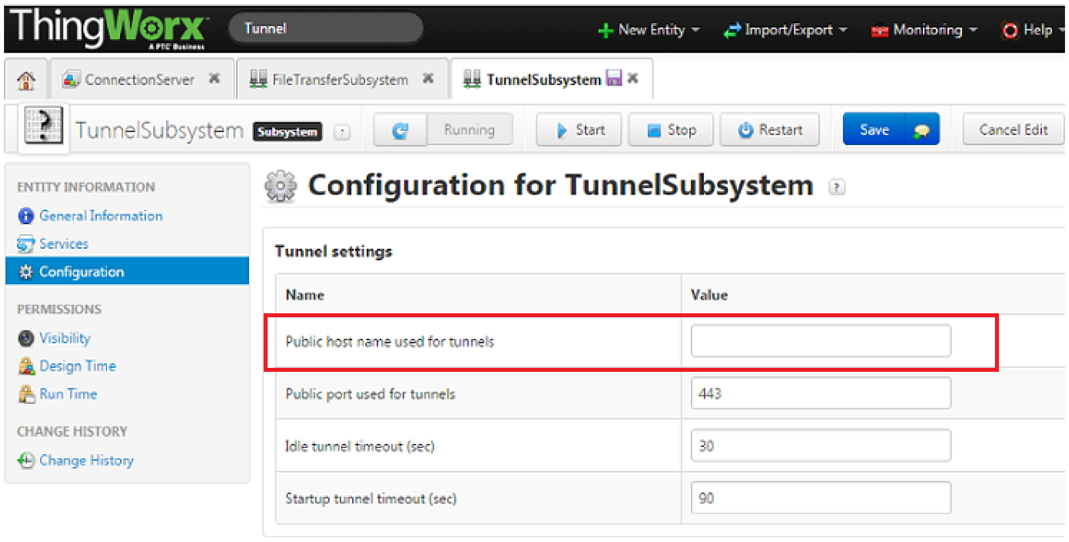
The following table lists and describes the parameters for tunneling:

Use	To Specify
`buffer_size`	The size of the buffer used for the tunnel, in bytes. The default value is 8192 bytes.
`read_timeout`	The time in milliseconds to wait for a WebSocket read before timing out.

Use	To Specify
	The default value is 10 milliseconds.
`idle_timeout`	The default idle timeout in milliseconds. The default value is 30000 milliseconds (30 seconds).
`max_concurrent`	The maximum number of concurrent tunnels. The default value is 4 tunnels.

Required Setting for the Tunneling Subsystem

When attempting to configure tunneling, you must check the configuration for the Tunneling Subsystem of ThingWorx Core. There is a field where you can specify the host/IP of the end point for the tunnel, called **Public host name used for tunnel**. The following figure shows the configuration page for the Tunneling Subsystem, with this field highlighted:



Configuration for TunnelSubsystem

Tunnel settings

Name	Value
Public host name used for tunnels	
Public port used for tunnels	443
Idle tunnel timeout (sec)	30
Startup tunnel timeout (sec)	90

Why do you need to configure this address? Suppose that you start up your server in Amazon EC2. The default IP address for the Tunneling Subsystem when the server is running in EC2 might be 10.128.0.x. Unless you change that address, the Tunneling Subsystem will tell the clients to attempt to connect to that host for the tunnel websocket. Since that IP address is a local network address, the tunnel will not work. Therefore, you must populate that configuration field with the external host/IP address that tunnels will use for connections.

Configuring File Transfers

To execute a file transfer, you need to configure options for both your client application and ThingWorx Core. Transfers can be executed in either direction: from the edge application to ThingWorx Core or from ThingWorx Core to the edge application.

Note

Keep in mind that the account associated with the Application Key must have the correct Read/Write permissions to the target and destination directories for a file transfer.

To transfer a file, the WS EMS must be configured with a set of virtual directories. The paths that you specify for the virtual directories must be absolute; the paths for the files must be relative to the virtual directories. For the WS EMS, the configuration section might look like this:

```
"file": {
  "virtual_dirs":[
      { "In" : "c:\Microserver_5.3.1-win32\microserver\in" },
      { “Out": "c:\Microserver_5.3.1-win32\microserver\out" },
      { "staging": "c:\Microserver_5.3.1-win32\microserver\staging" }
  ],
  "staging_dir": "staging"
}
```

If you use the additional parameters available for the `file` section, it might look like this:

```
"file": {
"buffer_size": 8192,
"max_file_size": 8000000000,
    "virtual_dirs":[
          { "In" : "c:\Microserver_5.3.1-win32\microserver\in" },
          { “Out": "c:\Microserver_5.3.1-win32\microserver\out" },
          { "staging": "c:\Microserver_5.3.1-win32\microserver\staging" }
     ]
     "staging_dir": "staging"
 }
```

This configuration is important because you must pass names of virtual directories in the parameters to the Copy service. As shown in the example above, you must use absolute paths.

The next example makes the following assumptions:

- A **RemoteThingWithFileTransfer** named RT1 exists on ThingWorx Core.
- The files are being transferred to/from the **SystemRepository** thing.
- The WS EMS is installed in the directory, `C:\Microserver` and that `C:\Microserver\in`, `C:\Microserver\out`, and `C:\Microserver\staging` exist.
- The source file is located in the `files` directory of the **SystemRepository** thing.
- The source and destination directories MUST exist AND be accessible to the WS EMS (Read/Write permissions).

In this example, the Copy parameters for a transfer from ThingWorx Core to the edge device would be:

- **sourceRepo**: `SystemRepository // Name of the Thing to transfer from`
- **targetRepo**: `RT1 // Name of the Thing to transfer to`
- **sourcePath**: `/files   // Directory in the SystemRepository` (absolute path)
- **targetPath**: `/In      // The name of a virtual dir`

  In this case, it is pointing to `C:\Microserver\in`. You can also specify subdirectories.
- **sourceFile**: `abc.json`
- **targetFile**: `abc.json  // Optional`
- The default is the **sourceFile** name. You can rename files during the transfer.

The paths on the WS EMS must be relative to a virtual directory that is registered to the remote thing (that is, they must start with the `"/<virtual_dir>"`). In the case of a file repository on ThingWorx Core, the paths need to be relative to the root directory of the file repository (must start with `"/"`).

Note that the things must be instances of one of the following templates:

- **FileRepository**
- **RemoteThingWithFileTransfer**
- **RemoteThingWithFileTransferAndTunneling**

Also, as of versions 5.0 and later of the WS EMS, you do not need the Lua Script Resource to do file transfers. You can add an `auto_bind` entry to your configuration file to specify the name of a thing that will participate in file transfers:

```
"auto_bind”: [
    { “name”: “RT1” }
   ]
```

The following table lists and describes the parameters for file transfers:

Use	To Specify
`buffer_size`	The size of the buffer used for the file transfer, in bytes. The default value is `8192` bytes.
`max_file_size`	The maximum size of a file that can be transferred, in bytes. The default value is `8000000000` bytes (8GB).
`virtual_dirs`	An array of virtual directories that are used when browsing and sending files to ThingWorx Core. The directories are defined using absolute paths, as shown in the example above.
`staging_dir`	A directory to use as a staging directory for files that will be transferred to the edge device.

Example Configurations

This section provides examples of how the WS EMS can be configured for several typical use cases.

Gateway Mode with Self-Identifying Remote Things Example

The WS EMS can be configured to run as a gateway, acting as the communication conduit and providing message relaying services for one or more remote things.

The WS EMS keeps a registry of the remote things it acts as a gateway for. You can set up the remote things to "self-identify" with the WS EMS. That is, when the remote things initialize and connect to the WS EMS, they send the information that uniquely identifies them to the WS EMS. This information is stored in the registry of the WS EMS.

For more information on configuring the `auto_bind` section of the configuration file, refer to the section, Configuring Automatic Binding for WS EMS on page 36 and to the section, AutoBound Gateways on page 37.

The example below illustrates how to configure the WS EMS for this scenario:

```
{
[{ "ws_servers":
"",                 "host":
443                 "port":
                 }],

"e34dbaf2-0984-4bfb-9a6c-4cbc57959763",   "appKey":

{  "ws_connection":
"ssl"                "encryption":
                    },

[{ "auto_bind":
"EdgeGateway001",   "name":
```

```
true                "gateway":
                 }]
}
```

Gateway Mode with Explicitly-Defined Remote Things Example

Although remote things can be set to identify themselves to the WS EMS, in many cases the remote things that connect to the WS EMS are well-known. In this case, you can explicitly define them within the configuration of the WS EMS.

For more information on configuring the `auto_bind` section of the configuration file, refer to the section, Configuring Automatic Binding for WS EMS on page 36 and to the section, AutoBound Gateways on page 37.

The example below illustrates how to configure the WS EMS for this scenario:

Note

In the example below, `EdgeThing001` is an explicitly defined remote thing that runs at the specified IP Address (`10.198.255.255`) and listens on port `8001`.

```
{
[{"ws_servers":
"",                 "host":
443                 "port":
                  }],
 "appKey":"e34dbaf2-0984-4bfb-9a6c-4cbc57959763",
 { "ws_connection":
 "ssl"               "encryption":
                  },
[{"auto_bind":
"EdgeGateway001",   "name":
true                "gateway":
                }, 
              {
"EdgeThing001",     "name":
                    "host": "10.198.255.255",
                    "port": 8001
               }]
}
```

Non-Gateway Mode with Self-Identifying Remote Things Example

The WS EMS can be configured to run in non-gateway mode, so things that attach to the WS EMS will pro-actively identify themselves with its process.

The example below illustrates how the WS EMS can be configured for this scenario:

```
{
 "ws_servers": {[
                "host": "localhost
                "port": 443
               }],
 "appKey":"xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx",
 "ws_connection": {
                   "encryption": "ssl"
                  },
"auto_bind": [{
               "name": "EdgeThing001",
               "host": "127.0.0.1",
               "port": 8001

             }]
}
```

Troubleshooting

This section discusses issues that can arise when using the WS EMS, along with recommended solutions.

Problem	**Possible Solution**
The WS EMS connects, but reports a time-out error when trying to authenticate.	Verify that you are running the required version of Tomcat. Refer to the *ThingWorx Edge Requirements and Compatibility Matrix*, which is available from the Reference Documentation page on the PTC Support site,
The WS EMS is failing to connect to my local server.	If your server is not configured to use HTTPS, set the `encryption` option of the WS EMS option to `none`. Before deployment, set the option back to `ssl`.

Problem	Possible Solution
I've started the WS EMS, made changes to config.json, but these changes are not reflected when I restart the WS EMS.	There is likely a syntax error (such as an extra comma, or similar) in your config.json. If the WS EMS is unable to start with the current config.json, it will use the last known good configuration file (config.json_booted). To verify that the problem is in the config.json, delete the config.json_booted file and restart the WS EMS. If it fails to start, check the config.json for errors.

The WS EMS connects to ThingWorx Core, authenticates successfully, but the thing I specified in the "auto_bind" section of my configuration file is not being created on ThingWorx Core.	The "auto_bind" section is an array of objects. Verify that you've enclosed the JSON object that represents your thing in square brackets as follows:
	`"auto_bind": [{`
	`"name": "RemoteThing001",`
	`"gateway": true`
	`}]`
	
	Instead of this, which would lead to this thing not being created on ThingWorx Core:
	`"auto_bind": {`
	`"name": "RemoteThing001",`
	`"gateway": true`
	`}`

Error Codes

This section lists and categorizes the error messages (and their codes) that can be returned.

General Errors

The following table lists general errors and their corresponding codes:

Code	Message	Troubleshooting
100	`TW_UNKNOWN_ERROR`	An error occurred, but it was not recognized. You should not see this error
101	`TW_INVALID_PARAM`	The parameter value is not allowed. This error message typically indicates that a NULL pointer was passed in. but a NULL pointer is not allowed for that parameter.

Code	Message	Troubleshooting
102	`TW_ERROR_ALLOCATING_MEMORY`	The amount of memory that was specified could not be allocated. Make sure that components free memory when they exit. Make sure that you free memory when your code no longer needs data structures. This error is very serious, and your application will usually terminate soon after.
103	`TW_ERROR_CREATING_MTX`	An error occurred while the WS EMS was creating a mutex.

WebSocket Errors

A system socket is used to run a WebSocket connection. A system socket sits one layer lower in the networking stack than a WebSocket. All WebSocket errors indicate some general issue communicating with ThingWorx Core. The following table lists WebSocket errors, their corresponding codes, and an explanation of the issue.

Code	Message	Troubleshooting
200	`TW_UNKNOWN_WEBSOCKET_ERROR`	An unknown error occurred on the WebSocket. You should not see this error.
201	`TW_ERROR_INITIALIZING_WEBSOCKET`	An error occurred while initializing the WebSocket. Check your WebSocket configuration parameters for validity.
202	`TW_TIMEOUT_INITIALIZING_WEBSOCKET`	A timeout occurred while initializing the WebSocket. Check the status of the connection to ThingWorx Core.
203	`TW_WEBSOCKET_NOT_CONNECTED`	The WebSocket is not connected to the Core instance. The requested operation cannot be performed.
204	`TW_ERROR_PARSING_WEBSOCKET_DATA`	An error occurred while parsing WebSocket data. The parser could not break down the data from the WebSocket.
205	`TW_ERROR_READING_FROM_WEBSOCKET`	An error occurred while reading data from the WebSocket. Retry the read operation. If necessary, resend the data.
206	`TW_WEBSOCKET_FRAME_TOO_LARGE`	The SDK is attempting to send a WebSocket frame that is too large. The Maximum Frame Size is set when calling `twAPI_Initialize` and should always be set to the Message Chunk Size (`twcfg.message_chunk_size`).
207	`TW_INVALID_WEBSOCKET_FRAME_TYPE`	The type of the frame coming in over the WebSocket is invalid.
208	`TW_WEBSOCKET_MSG_TOO_LARGE`	The application is attempting to send a message that has been broken up in to chunks that are too large to fit in a frame. You should not see this error.

Code	Message	Troubleshooting
209	TW_ERROR_WRITING_TO_ WEBSOCKET	An error occurred while writing to the WebSocket.
210	TW_INVALID_ACCEPT_KEY	The Accept key sent earlier from ThingWorx Core is not valid.

Messaging Errors

The following table lists the error codes and messages for Messaging errors and provides some troubleshooting information.

Code	Message	Troubleshooting
300	TW_NULL_OR_INVALID_MSG_HANDLER	The message handler singleton has not been initialized.
301	TW_INVALID_CALLBACK_STRUCT	The callback structure was not valid. Check that your application properly implements the callback.
302	TW_ERROR_CALLBACK_NOT_FOUND	The specified callback was not found. Check the callback parameters passed to the function.
303	TW_INVALID_MSG_CODE	An attempt to set an invalid message code was made. You should not see this internal error in your code.
304	TW_INVALID_MSG_TYPE	A function was called with an invalid message code. You should not see this internal error.
305	TW_ERROR_SENDING_MSG	An error occurred while sending the message. Check the network connections and the destination host. If network connections and the destination host are working properly, check the configuration of the destination host to be sure it is correct.
306	TW_ERROR_WRITING_OFFLINE_MSG_ STORE	An error occurred while writing to the offline message store.
307	TW_ERROR_MESSAGE_TOO_LARGE	The message was too large. Check that the size you configured for messages is adequate for all expected traffic. Consider increasing the size.
308	TW_WROTE_TO_OFFLINE_MSG_STORE	The message was not sent to ThingWorx Core, but was stored in the offline message store. The message will be delivered next time the WebSocket is connected.
309	TW_INVALID_MSG_STORE_DIR	The directory for the message store was not correct. Make sure the path is valid and that you have write permission.
310	TW_MSG_STORE_FILE_NOT_EMPTY	The on-disk file that is uses to store offline messages contains some messages that have not been sent yet. The file name cannot be changed.

Primitive and InfoTable Errors

The following table lists the errors related to the data structures, `twPrimitive` and `twInfoTable`, and their supporting functions. It also provides suggestions for troubleshooting.

Code	Message	Troubleshooting
400	`TW_ERROR_ADDING_DATASHAPE_ENTRY`	An error occurred while attempting to add an entry (field) to the data shape.
401	`TW_INDEX_NOT_FOUND`	Attempted to access a non-existent field from a row in an infotable. The index value must be less than the number of fields defined in the data shape.
402	`TW_ERROR_GETTING_PRIMITIVE`	The function `twInfoTable_GetPrimitive` failed to retrieve the requested primitive from the infotable.
403	`TW_INVALID_BASE_TYPE`	The specified base type is not valid. Check the spelling in your code, or select a different base type.

List Errors

The following table lists the error related to lists (for example, subscribed properties):

Code	Message	Troubleshooting
500	`TW_LIST_ENTRY_NOT_FOUND`	The entry was not found in the list. For example, the requested property was not found in the list of subscribed properties.

API Errors

The following table lists the errors related to the API:

Code	Message	Troubleshooting
600	`TW_NULL_OR_INVALID_API_SINGLETON`	The API singleton is either null or invalid. This error occurs if the API was not initialized properly. Check the parameters that you are passing to the initialize function. Check the log.
601	`TW_ERROR_SENDING_RESP`	An error occurred while sending a response message to ThingWorx Core.
602	`TW_INVALID_MSG_BODY`	A message was received from ThingWorx Core that had an invalid or malformed message body.
603	`TW_INVALID_MSG_PARAMS`	A Property PUT was received from ThingWorx Core with an infotable that contains empty parameters. The property value will not be changed.
604	`TW_INVALID_RESP_MSG`	The response message was not valid. You should not see this internal error.
605	`TW_NULL_API_SINGLETON`	The API singleton was null. This message

Code	Message	Troubleshooting
		indicates that the API was not initialized properly. Check the parameters that you are passing to the initialize function. Check the log.
606	`TW_ERROR_CREATING_MSG`	An error occurred while creating the message. This error typically indicates an out-of-memory condition.
607	`TW_ERROR_INITIALIZING_API`	An error occurred while initializing the API. Check the parameters that you are passing to the initialize function. Check the log.

Tasker Errors

The following table lists the errors related to the Tasker:

Code	Message	Troubleshooting
700	`TW_MAX_TASKS_EXCEEDED`	You have attempted to create more tasks than are allowed for the built-in tasker. If you have many tasks running you may wish to consider using native threads if your platform supports them.
701	`TW_TASK_NOT_FOUND`	The specified task ID was not found. Make sure the task ID passed to this function is correct.

Logging Errors

The following table lists the error related to logging:

Code	Message	Troubleshooting
800	`TW_NULL_OR_INVALID_LOGGER_SINGLETON`	The logger singleton was not initialized properly. This error indicates a memory allocation error.

Utils Errors

Base64 encoding/decoding is used. The following table lists the related errors. At this time, the code does not use them.

Code	Message
900	`TW_BASE64_ENCODE_OVERRUN`
901	`TW_BASE64_DECODE_OVERRUN`

System Socket Errors

System Sockets are Operating System-provided networking APIs. The `TW_ERROR_WRITING_TO_SOCKET` error in the System Socket category is a general socket write error. All errors in this category are in the context of a connection to ThingWorx Core.

As appropriate, first check the network connection between the thing where your application is running and ThingWorx Core to resolve the problem. If a proxy server is used between your thing and ThingWorx Core, check that the proxy server is operating properly. If so, check the configuration for the connection to the proxy server.

Code	Message	Troubleshooting
1000	`TW_ERROR_WRITING_TO_SOCKET`	General socket write error encountered while writing to ThingWorx Core.
1001	`TW_SOCKET_INIT_ERROR`	An error occurred while initializing the socket. The network connection may have dropped.
1002	`TW_INVALID_SSL_CERT`	The SSL certificate provided by the server was not valid or was self-signed. Check your certificate settings.
1003	`TW_SOCKET_NOT_FOUND`	The socket was not found. The network connection may have dropped.
1004	`TW_HOST_NOT_FOUND`	The specified Core instance was not found. Check network connections. Ensure that your application configuration specifies a valid host address.
1005	`TW_ERROR_CREATING_SSL_CTX`	An error occurred creating the SSL context.
1006	`TW_ERROR_CONNECTING_TO_PROXY`	An error occurred connecting to the specified proxy server. Make sure the proxy server address is correctly specified. Check network connections.
1007	`TW_TIMEOUT_READING_FROM_SOCKET`	An attempt to read from a socket timed out with no data available.
1008	`TW_ERROR_READING_RESPONSE`	An error occurred while reading the response from the proxy server. Check your proxy configuration in your application.
1009	`TW_INVALID_PROXY_CREDENTIALS`	The credentials presented to the proxy server were not valid. Check with the administrator for the proxy server and re-enter the credentials for the proxy server. NOTE: While the connection to the proxy server is not encrypted, the credentials are obfuscated using standard HTTP Basic, Digest, or NTLM encoding.
1010	`TW_UNSUPPORTED_PROXY_AUTH_TYPE`	The specified authentication type for the proxy server is not supported. Make sure that the authentication type is correctly specified in your application.

Code	Message	Troubleshooting
1011	`TW_ENABLE_FIPS_MODE_FAILED`	FIPS Mode could not be enabled. Ensure that you are using an OpenSSL library with FIPS validated cryptographic algorithms.
1012	`TW_FIPS_MODE_NOT_SUPPORTED`	FIPS Mode is not supported. Ensure that you are using an OpenSSL library with FIPS validated cryptographic algorithms.

Message Code Errors

The message code errors can be returned when the SDK makes a request to ThingWorx Core. They can also be the return values for property/service requests executed by the application using the SDK. For example, if the server queried the SDK application for the property 'temperature', but the application did not have that property, it could return `TW_NOT_FOUND`. The server could also return the same code if the application asked the server for a property that it did not have defined.

Most of these are standard HTTP error codes. You can see more information about them at http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html.

Code	Message	Troubleshooting
1100	`TW_BAD_REQUEST`	The HTTP request contained syntax errors, so the server did not understand it. Modify the request before attempting it again..
1101	`TW_UNAUTHORIZED`	The request requires authentication. This error results from a failed login attempt — whether from credentials that were not valid or from the request being sent before authentication occurred.
1102	`TW_ERROR_BAD_OPTION`	An option or a parameter for a function has a value that is not valid or is not spelled correctly (and so is not recognized).
1103	`TW_FORBIDDEN`	ThingWorx Core is denying you access to the requested resource. Check your permission settings on ThingWorx Core.
1104	`TW_NOT_FOUND`	This message is returned for anything that was not found — a property, a service, a thing, a data shape, and so on.
1105	`TW_METHOD_NOT_ALLOWED`	The specified method is not allowed. Check the spelling and syntax of your code.
1106	`TW_NOT_ACCEPTABLE`	Not acceptable.
1107	`TW_PRECONDITION_FAILED`	The precondition for the operation was not met.

Code	Message	Troubleshooting
1108	`TW_ENTITY_TOO_LARGE`	This error occurs if you attempt to send a property, or service or event parameters that are too large for ThingWorx Core to handle.
1109	`TW_UNSUPPORTED_CONTENT_FORMAT`	This error occurs if you attempt to send a property, or service or event parameter that has the wrong base type as defined on ThingWorx Core.
1110	`TW_INTERNAL_SERVER_ERROR`	An error occurred on ThingWorx Core while processing this request.
1111	`TW_NOT_IMPLEMENTED`	ThingWorx Core may return this error if you attempt a function that is not implemented.
1112	`TW_BAD_GATEWAY`	A gateway could be bad if it cannot communicate to the next component in the chain.
1113	`TW_SERVICE_UNAVAILABLE`	The requested service is not defined. You could also use the `TW_NOT_FOUND` error code, but this one is more specific.
1114	`TW_GATEWAY_TIMEOUT`	If the application sends a request to ThingWorx Core and does not get a response within some amount of time, the service call results in this error. You can configure the amount of time.

Subscribed Properties Errors

The following table lists the errors related to subscribed properties:

Code	Message	Troubleshooting
1200	`TW_SUBSCRIBEDPROP_MGR_NOT_INTIALIZED`	The Subscribed Properties Manager has not been initialized. The Subscribed Properties Manager is initialized when `twApi_Initialize` is called only if `ENABLE_SUBSCRIBED_PROPS` is defined. If you wish to use this functionality make sure `ENABLE_SUBSCRIBED_PROPS` is defined.
1201	`TW_SUBSCRIBED_PROPERTY_NOT_FOUND`	The requested subscribed property was not found.

File Transfer Errors

The following table lists the errors for the File Transfer component:

Code	Message	Troubleshooting
1300	`TW_FILE_XFER_MANAGER_NOT_INITIALIZED`	The File Transfer Manager has not been initialized. The File Transfer Manager is initialized when `twApi_Initialize` is called only if `ENABLE_FILE_XFER` is defined. If you wish to use file transfer functionality make sure `ENABLE_FILE_XFER` is defined.
1301	`TW_ERROR_CREATING_STAGING_DIR`	An error occurred while creating the staging directory. The error happens if there is an invalid path or if you do not have the proper permissions to create the directory specified.
1302	`TW_FILE_NOT_FOUND`	The specified file for the transfer was not found. Check the name of the file specified. If it is correct, check for the presence of the file in the file system at the specified location.
1303	`FILE_TRANSFER_FAILED`	The file transfer operation failed. The network connection may have dropped during the transfer, the destination for the transfer may be unavailable (down for maintenance or power outage), or the MD5 checksum of the file indicated invalid file content.

Tunneling Errors

The following table lists the errors related to the Tunneling Manager

Code	Message	Troubleshooting
1400	`TW_TUNNEL_MANAGER_NOT_INITIALIZED`	The Tunnel Manager has not been initialized. The Tunnel Manager is initialized when `twApi_Initialize` is called only if `ENABLE_TUNNELING` is defined. If you wish to use tunneling functionality make sure `ENABLE_TUNNELING` is defined.
1401	`TW_TUNNEL_CREATION_FAILED`	The tunnel was not created. This error could be because of an out-of-memory condition.

REST APIs and WS EMS

REST (Representational State Transfer) is an important communication tool that provides much of the communication functionality that Web Services are used for, but without many of the complexities. As a result, REST is much easier to work with and can be used by any client that is capable of making an HTTP request.

Note

The examples in this topic assume that you are familiar with executing HTTP **POST** methods in your web development environment or application.

URL Pattern

The following URL pattern is used when communicating with ThingWorx Core:

```
<http|https>://<host>:<port>/Thingworx/<entity collection>/
<entity>/<characteristic collection>/<characteristic>?<query parameters>
```

You can use this URL pattern when you want to access information that is stored on ThingWorx Core for a remote device or machine that is running a WS EMS. The pattern to use when you want to access the WS EMS that is running on a remote device or machine is similar:

```
<http|https>://<host>:<port>/Thingworx/<entity collection>/
<entity>/<characteristic collection>/<characteristic>?<query parameters>
```

Note

Anything enclosed in angle brackets (< >) is information that you may need to provide. Some user-supplied information in the URLs above is required, while some is optional. The information that is required depends on the type of request that is being made.

Example

The following API call executes the service **GetLogData** that is associated with the thing called `ACMElocking_valve`.

```
http://localhost/Thingworx/Things/ACMElocking_valve/Services/GetLogData
```

The REST method is GET for this example. To view the returned data in JSON format, select the value `application-json` for the Accept Header in the REST client.

Built-in Collection Values

ThingWorx has a finite list of entity collections. Each entity collection contains entities (for example, `Things`) of the respective type (for example, `/Things` contains all things). The WS EMS supports the `Things` entity collection and the `Properties` and `ThingName` characteristic collections. It also supports the File Transfer Subsystem and Services that are associated with Things. For more information, refer to the section. REST APIs Supported by WS EMS on page 66.

Updating, Deleting, and Executing with REST APIs

The following rules help to understand what is needed based on the type of request being made.

	Notes	Sample URL	HTTP Action	Content Type
UPDATE	Updates require specifying the entity part (“`thing_name`” in the sample)	`http://host/Thingworx/Things/thing_name/`	`PUT`	application/json or text/xml
DELETE	Deletes require specifying the entity part as well	`http://host/Thingworx/Things/thing_name`	`DELETE`	n/a
INVOKING SERVICES	Calling a service requires specifying the complete URL, including the specific characteristic	`http://host/Thingworx/Things/MyThing/Services/myService` If your service requires them, these inputs should be passed in the form fields of your `POST`.	`POST`	application/json

Executing HTTP Requests

When executing HTTP requests, use UTF-8 encoding, and specify the optional port value if required.

Best Practice

Use HTTPS in production or any time network integrity is in question.

Handling HTTP Response Codes

In most cases, you should expect to get back either content or the status code of 200, which indicates that the operation was successful. In the case of an error, you receive an error message.

Working with HTTP Content

If you are sending or receiving any HTTP content (JSON, XML, HTML [for responses only]), set the request content-type header to the appropriate value based on the HTTP content you are sending. The following table lists and briefly describes the HTTP methods that are supported:

Supported HTTP Methods

Use	To
GET	Retrieve a value.
PUT	Write a value or create new things or properties.
POST	Execute a service.
DELETE	Delete a thing or property.

For Content Type	In Accept Header, Use
JSON	application/json
XML	text/xml
HTML	text/html (or omit Accept header)

Metadata

You can display the metadata of any specific thing, thing template, or data shape you build by going to the following URL in a web browser: `NameoftheThing/Metadata`

Note

To see it, this information must be shown as JSON.

Passing in Authentication with your REST API Call

To authenticate with ThingWorx Core for a REST API call, use an Application Key that is associated with a user account that has the privileges to perform the actions that you intend to invoke, using the REST APIs. For REST calls to a WS EMS, you do not need authentication

Best Practice

Although you can pass in a User name and Password with your REST call, the recommended best practice is to use an Application Key. Generate the key in ThingWorx Composer, and then pass it with your REST call. The user account associated with the Application Key should have privileges to read/write properties and run services on the related devices/machines in ThingWorx Core.

Reading and Writing Properties Using the REST API

Reading a Property Value

To read a property from the local WS EMS, you can use a **GET** from the REST client and the following URL:

```
http://localhost:8000/ThingWorx/Things/thing_name/Properties/prop_name
```

Notice that you are pointing at the local WS EMS, on port 8000, to retrieve a description. By default, WS EMS listens on port 8000. Also by default, the WS EMS accepts requests only from an application that is running on the same machine as it is (i.e., `localhost`). You can configure a WS EMS to accept requests from other IP addresses.

When you execute this request, the WS EMS pushes it to ThingWorx Core. Keep in mind that the WS EMS has no state. It does not even know that the property exists. It just takes the request URL, breaks it up and repackages it, translates it into the AlwaysOn protocol, and forwards it to ThingWorx Core, which responds with its current value for that property. The result type is always of base type `INFOTABLE`, with the property name and current value.

Tip

To debug a problem with a property not updating or a service not executing, set the `level`and `publish_level`parameters (in the `logger`section of the `config.json` file) to `TRACE` and in the `ws_connection` section, set the `verbose`parameter to `true`. That way, you can see all the activity passing between the WS EMS and ThingWorx Core.

For example, if the response seems to be returned slowly, by logging at the `TRACE` level and setting `verbose` to true, you can check the timestamps for the request and response to calculate the actual time. To match a request with a response, locate the `Request ID` of the outgoing message and the `Request ID` included in the incoming response message.

Writing a Property Value

To write a property value to ThingWorx Core through a WS EMS for an edge device managed by a Lua Script Resource, select the **PUT** method in the REST client and use the same URL as a read (**GET**) for the property. For example:

```
http://localhost:8000/ThingWorx/things/thing_name/Properties/<prop_name>
```

Then, in the area provided in the client, enter the property name and value, using JSON format:

```
{ "<prop_name>" : "Hello World from ThingWorx" }
```

It is important to remember that ThingWorx Core recognizes the **PUT** as coming from the edge device and updates the value for the device. ThingWorx Core does not attempt to write it to the device.

If you shut down the Lua Script Resource and execute the same **PUT**, the value is written to ThingWorx Core for the device that is running WS EMS rather than the LSR device. The distinction is between writing the value directly to ThingWorx Core as opposed to writing the value through the WS EMS. In both instances you see the value on ThingWorx Core.

You also need to set the `Content-type` for a write to the format you are using. In this case, it is `application/json`. If the device for the property is a remote thing, the property is also remote. If that device is not bound, you cannot write the value to the property. If the device is connected through a WS EMS and Lua Script Resource and the WS EMS is running, you can start up the Lua Script Resource that is configured for the remote thing. Once the remote thing is bound, ThingWorx Core can send the write request to the WS EMS.

Note that ThingWorx Core does not change the property value until the request has made the full round trip:

1. A request is sent from a REST client to ThingWorx Core.
2. ThingWorx Core recognizes it as a remote property and forwards to the remote thing.
3. If the remote thing is running a WS EMS, the WS EMS sends it to LSR, which writes it internally.

   At this point, the in-memory value for ThingWorx Core is still the old value. The LSR should be set up to send the value back up to ThingWorx Core.
4. Only after the LSR sends the new value back to ThingWorx Core does the value change there.

Transferring Files through the REST API

To prepare for file transfers, set up the WS EMS with virtual directories to send and receive files. If it is not already running, start the LSR for the device so that the virtual thing at the edge is up and running and ThingWorx Core knows it is connected.

To invoke a file transfer from ThingWorx Core, use the HTTP POST method and the following URL:

```
http://<server_name>:<port>/ThingWorx/Subsystems/FileTransferSubsystem/
        Services/Copy
```

In the area provided in the REST client, enter the parameters for the `Copy` service (in a JSON object), as indicated here:

```
{
  "sourceRepo" : "<Enter a Valid Repository"
  "sourcePath" : "<Enter a Valid Path>"
  "sourceFile" : "<Enter a Valid File>"
  "targetRepo" : "<Enter a Valid Thing>"
  "targetPath" : "<Enter a Valid Path>"
  "targetFile" : "<Enter a New Name for the file (optional>"
}
```

The parameters are broken down by target and source (you can view the parameters by looking at the definition for the Copy service in ThingWorx Composer).

Note

Refer to ThingWorx Core Help Center for more information about the Copy services, specifically the details concerning what is a valid file repository (that is, which templates support file transfer). ThingWorx Core Help Centers for different releases are available from the PTC Help Centers page on the PTC Support site.

When you run the request, you can see the results (in JSON) format in the REST client. Scroll down until you see the rows of the infotable. The value of the `state` parameter is `"validated"` if the file was transferred successfully.

You can also execute a file transfer through the WS EMS, using the same parameters and same POST, with the URL pointed at the local WS EMS:

```
http://localhost:8000/ThingWorx/Subsystems/FileTransferSubsystem/
        Services/Copy
```

The headers for the WS EMS differ in that you have only the `content-type`header for the WS EMS. The results are the same (except that the WS EMS puts the rows at the top and the data shape at the bottom).

Why use the WS EMS or an SDK with the REST API instead of just calling ThingWorx Core REST APIs from an application? There are some benefits to using the WS EMS or an SDK just to interact with ThingWorx Core, using the REST APIs:

- You can have a secure connection when you use a WS EMS or an SDK to interact with ThingWorx Core.
- The AlwaysOn protocol persists the connection between an application and ThingWorx Core.
- When a WS EMS or an SDK makes the REST calls instead of your application, you save a lot in terms of resource usage on ThingWorx Core could potentially have to handle hundreds of HTTP requests coming from an application that is running on hundreds of devices, all sending multiple requests. Typically, the most expensive part of HTTP request is opening the socket — all the headers that are sent across the wire and so forth.

When you use a WS EMS or SDK, you eliminate the burden on ThingWorx Core. The WebSocket connection is already set up (and persisted), and the multiple requests for an application are sent over the single WebSocket. In addition, WS EMS and the SDKs send the requests using binary data, which results in more efficient use of bandwidth (in terms of the number of bytes that go across the wire).

- With the WS EMS or an SDK, the step to set up the socket is eliminated. Requests and responses can be exchanged more quickly. Especially if you have multiple applications that are making multiple requests behind a WS EMS, performance improvements are significant when you use the WS EMS to pass REST requests instead of passing them directly to ThingWorx Core.

Configuring WS EMS to Listen on IP Other Than localhost

You can configure a WS EMS to listen on a specific IP address or on all IP addresses available on the device, using the `http_server` section of the configuration file. If a device has one network card and it is configured with an IP address of 11.128.0.3, the device effectively has two IP addresses, 11.128.0.3 and 127.0.0.1 (localhost). To configure the specific IP address, set the `host` parameter, as follows:

```
"http_server" : {
  "host": "11.128.0.3",
  "port": 9010
}
```

In this configuration, any application must use the specific IP address to access the device. “localhost” does not work.

If a device has two network cards with corresponding IP addresses for the device and you want users to access the device using any of the IP addresses available, use `0.0.0.0` for the host IP address, as follows:

```
"http_server" : {
  "host": “0.0.0.0”,
  "port": 9010
}
```

In either configuration, you can leave the port number as is or change it.

It is important to keep in mind that these configurations expose the REST interface to any client on the network that wants to access the WS EMS. To provide secure access, configure a user name and password for the HTTP server as well as SSL, as shown here:

```
"http_server" : {
  "user": "acmeAdmin”,
  "password": "AES:EncryptedPassword",
  "ssl": true
}
```

REST APIs Supported by WS EMS

The WS EMS is built to reflect the REST API of ThingWorx Core, but it is not a complete reflection. Rather, it is reflection of those APIs that are most useful at the Edge. For example, if you try to look at Resources, nothing is returned. If you try to look at properties for a thing that either is running the WS EMS or that is registered with it (WS EMS is running as a gateway), you can see all the properties. By default, the WS EMS returns data in JSON format.

For example, these APIs do work with a WS EMS:

- **/Thingworx/Things/thing_name/Properties**
- **/Thingworx/Things/thing_name/Properties/prop_name**

where **thing_name** represents the name of any device that is connected to the local WS EMS where you are using a REST API, and **prop_name** represents the name of any property for the specified device.

Browser-based Use of REST APIs

The REST APIs of ThingWorx Core can be used in a browser or an application that supports the HTTP commands. However, the behavior of the REST APIs on a WS EMS is slightly different.

You cannot use a **PUT** through a query parameter of the REST API in a browser, as on ThingWorx Core. For example, the following use of **PUT** works on ThingWorx Core but not on a WS EMS:

```
https://<server_ip/ThingWorx/Things/<thing_name>/Properties/<property_name>/
        method=put&<prop_name>=<value>
```

You actually must do an HTTP **PUT** to the WS EMS, using a REST client that can do an HTTP **PUT**.

Another difference with ThingWorx Core lies in how WS EMS returns the information for a specific property. Both ThingWorx Core and WS EMS use an infotable to return the information. However, WS EMS returns the rows first and the data shape second. This order is the opposite from the order in which ThingWorx Core returns the information.

Similarly, for services, the following use of a service such as **GetDescription** works on ThingWorx Core but not on a WS EMS:

```
https://<server_ip>/ThingWorx/Things/thing_name/Services/GetDescription
```

For WS EMS, you must request to run services by using a **POST** through a client that can do an HTTP **POST**. You could do it this way from a browser, as long as you do not have any input parameters for the service:

```
https://localhost:8000/ThingWorx/Things/thing_name/Services/
        GetDescription/method=POST
```

However, if you have parameters, use a client that can do an HTTP **POST**.

Here are a few of the REST APIs of ThingWorx Core that do not work with WS EMS:

- **/Thingworx/Things**
- **/Thingworx/Things/thing_name**
- **/Thingworx/Resources**

By default, a WS EMS uses `application/json` for the `Accept` header.

Using a REST Client with a WS EMS

You can use a REST client such as Postman to run REST APIs against a WS EMS. You can save them and even set up collections of them. In addition, you can export a collection and import them into a javascript engine such as Node.js.

When you are using REST APIs, it is important to set your headers correctly:

- `Accept`— Specifies the format in which the data should be returned — JSON for WS EMS. For ThingWorx Core, you can also choose XML or text.
- Application Key (`appKey`) — Provides the authentication you need with ThingWorx Core. You do not need it when running a REST API against a local WS EMS.

  You can also use basic authentication. In Postman, you can select **Basic Authorization** and specify a user name and password to access ThingWorx Core.
- `x-thingworx-session` — Determines if your request will set up an HTTP session with ThingWorx Core. Having a session makes it possible to send multiple requests from a browser to ThingWorx Composer without having to authenticate with each request. When you set up a session, the browser and Composer authenticate each request in the background.

  However, in an application, you do not want a session because sessions take up memory. For an application, set this header to `false`so that ThingWorx Core does not create a session every time that the application sends a request. Sending the `appKey` with each request does not impact memory.

Note

If you use Basic authentication, you always get a session with a ThingWorx Core instance. With an application, use an `appKey` for authentication and set the `x-thingworx-session` header to `false`.

Using Services with a WS EMS

The following services work as REST APIs with both ThingWorx Core and a WS EMS:

- AddEdgeThing on page 68
- GetConfiguration on page 69
- GetEdgeThings on page 69
- GetLogData on page 70
- GetMicroserverVersion on page 70
- HasEdgeThing on page 71
- RemoveEdgeThing on page 71
- ReplaceConfiguration on page 72
- Restart on page 73
- StartFileLogging on page 73
- StopFileLogging on page 74
- TestPort on page 75
- UpdateConfiguration on page 75

The WS EMS also supports using the `isConnected` property with a REST API.

The rest of this section provides details about these services.

AddEdgeThing

The **AddEdgeThing** service adds an edge device to the devices that are currently connected to the WS EMS.

Inputs

Pass in `TW_INFOTABLE` that has one row and two columns. The row object must contain the following parameters:

- The `name`of the device (thing) that you want to add. Keep in mind that the device must exist on ThingWorx Core as a thing that was created with the **RemoteThing** thing template. For example: `{"name":"NameOfThing"}`
- `persist`, which specifies whether the device should be added to the list of devices that are automatically bound to the corresponding thing on ThingWorx Core. The format for this parameter is `"persist":true|false`.

Outputs

This service returns an HTTP response only. If the operation was successful, it returns HTTP 200. Otherwise, it returns an HTTP error.

Example

Here is an example of a REST call that adds an Edge thing:

```
http://localhost:8000/Thingworx/Things/LocalEms/Services/
    AddEdgeThing{"name" : "acmeEdgeThing1"}"persist":true
```

GetConfiguration

The **GetConfiguration** service retrieves the configuration that the WS EMS is currently using

Note

This service does not return the current `config.json` file. Rather it returns the configuration that is currently loaded into the WS EMS. For example, if you call **UpdateConfiguration** but do not restart the WS EMS, the **GetConfiguration** service returns the configuration parameters and their values that the WS EMS is using, not the `config.json` file. You do not see the changes that were passed in with **UpdateConfiguration**.

Inputs

This service does not take any input parameters.

Outputs

This service returns a `TW_INFOTABLE` that contains a json object. The object contains the configuration parameter/value pairs that are currently loaded in the WS EMS.

Example

Here is an example of a REST call that retrieves the configuration of a WS EMS:

```
http://localhost:8000/Thingworx/Things/LocalEms/Services/GetConfiguration
```

GetEdgeThings

The **GetEdgeThings** service returns a list of edge things that are registered with the WS EMS gateway.

Inputs

The name of the WS EMS gateway.

Outputs

This service returns a `TW_INFOTABLE` that contains the names of the Edge things that are registered with the WS EMS gateway.

Example

Here is an example of the REST call that retrieves the names of the Edge things that are registered with the WS EMS gateway:

```
http://localhost:8000/Thingworx/Things/LocalEms/Services/GetEdgeThings
```

GetLogData

The **GetLogData** service retrieves the log entries from the WS EMS.

Inputs

Pass in a `TW_INFOTABLE` that contains one row and three columns. The row object must contain the following parameters;

- `startDate` - The oldest log entry to retrieve (as a `DATETIME`).
- `endDate` - The newest log entry to retrieve (as a `DATETIME`).
- `maxItems` - Max number of entries to retrieve (as an `INTEGER`).

Outputs

This service returns a `TW_INFOTABLE` that contains the log entries. (The related data shape is `logEntry`.)

Example

Here is an example of a REST call that retrieves log entries for a WS EMS:

```
http://localhost:8000/Thingworx/Things/LocalEms/Services/
  GetLogData{"startDate":"080115", "endDate":"081515", "maxItems":50}
```

GetMicroserverVersion

The **GetMicroserverVersion** service returns the version of the WS EMS.

Inputs

None

Outputs

This service returns a string that contains the version of the WS EMS. For example, `5.3.2`

.

Example

Here is an example of a REST call that retrieves the version of a WS EMS:

```
http://localhost:8000/Thingworx/Things/LocalEms/Services/GetMicroserverVersion
```

HasEdgeThing

The **HasEdgeThing** service checks whether a certain thing is connected to the WS EMS.

Inputs

You can pass in raw JSON or an infotable that contains the name of the thing whose connection you want to check:

- `TW_INFOTABLE`, that contains the name of the thing to check. The infotable can be passed in as raw json. For example, `{"name":"ThingName"}`

Outputs

- `TW_INFOTABLE`, which contains the result, as `TW_BOOLEAN`. Returns `true` if the specified thing is connected, `false` otherwise.

Example

Here is an example of a REST call that determines if a specified edge thing is connected to a WS EMS:

```
http://localhost:8000/Thingworx/Things/LocalEms/Services/
    HasEdgeThing{"name":"acmeEdgeThing1"}
```

RemoveEdgeThing

The **RemoveEdgeThing** service removes the specified device from the set of devices that are connected to the WS EMS. In addition, it removes the device from the list of devices that should bind automatically when the WS EMS contacts ThingWorx Core.

Inputs

You pass in an infotable that contains the name of the thing that you want to remove from the WS EMS:

- `TW_INFOTABLE`, that contains the name of the thing to remove. The infotable can be passed in as raw json. For example, `{"name" : "ThingName"}`

Note

If the removal is permanent, make sure that you also delete the thing on ThingWorx Core side.

Outputs

This service returns an HTTP response only. If the operation was successful, it returns HTTP 200. Otherwise, it returns an HTTP error.

Example

Here is an example of a REST call that deletes an Edge thing from the list of devices that should bind automatically when the WS EMS contacts ThingWorx Core:

```
http://localhost:8000/Thingworx/Things/LocalEms/Services/
    RemoveEdgeThing{"name" : "acmeEdgeThing1"}
```

ReplaceConfiguration

The **ReplaceConfiguration** service allows you to replace the configuration file for the WS EMS (`config.json`).

Tip

The new configuration file does not take effect until you restart the WS EMS. Use the Restart on page 73 service to force the changes to take effect.

Inputs

Pass in a `TW_INFOTABLE` that contains a JSON object. This object must contain

- `"config"` — A JSON string that is used to replace the current configuration file.

Outputs

This service returns an HTTP response only. If the operation was successful, it returns HTTP 200. Otherwise, it returns an HTTP error.

Example

Here is an example of a REST call that replaces the configuration of a WS EMS that is running on your computer, using this service:

```
http://localhost:8000/Thingworx/Things/LocalEms/Services/
    ReplaceConfiguration{"ws_servers"["host":newServer.acme.com",
        "port":80,appKey="<encrypted_application_key>"]
        "certificates"["disableCertValidation":true]}
```

Restart

The **Restart** service restarts the WS EMS.

Inputs

Pass in the following parameter:

- The `name` of the device (thing) that you want to restart. For example: `{"name":"NameOfThing"}`

Outputs

This service returns an HTTP response only. If the operation was successful, it returns HTTP 200. Otherwise, it returns an HTTP error.

Example

Here is an example of a REST call that restarts the WS EMS:

```
http://localhost:8000/Thingworx/Things/LocalEms/Services/Restart
```

StartFileLogging

The **StartFileLogging** service tells the WS EMS to begin writing log messages to a file. The WS EMS generates log messages at the level defined by a call to this service.

Inputs

You pass in `TW_INFOTABLE` that has one row and two columns. The row object must contain the following parameters:

- The `level` of the log messages to write to a file. Choose among the following levels: `TRACE`, `DEBUG`, `WARN`, `INFO`, or `AUDIT`, where `TRACE` provides the most information.
- `directory`, which specifies the path to the file on the computer where WS EMS is running. Use the following format: `/twx/wsems/logfiles/directory`.

Note

The `TRACE` level is useful when testing and troubleshooting. It provides both the operational and functional log messages for a WS EMS.

Outputs

This service returns an HTTP response only. If the operation was successful, it returns HTTP 200. Otherwise, it returns an HTTP error.

Example

Here is an example of a REST call that tells a WS EMS to start logging messages that it generates to a file, with the log level set to TRACE:

```
http://localhost:8000/Thingworx/Things/LocalEms/Services/
    StartFileLogging{"level":"TRACE","directory":"./ws_ems/logfiles"}
```

StopFileLogging

The **StopFileLogging** service tells the WS EMS to stop writing log messages to a file.

Inputs

You pass in the following parameter:

- `"delete" ; true | false`. Set to `true` to stop the logging to a file, or `false` to continue logging to a file.

Outputs

This service returns an HTTP response only. If the operation was successful, it returns HTTP 200. Otherwise, it returns an HTTP error.

Example

Here is an example of a REST call that tells a WS EMS to stop logging messages that it generates to a file and to delete the existing log file:

```
http://localhost:8000/Thingworx/Things/LocalEms/Services/StopFileLogging&delete=true
```

TestPort

The **TestPort** service tests the connection to a specified ThingWorx Core instance and port.

Inputs

Pass in the following parameters:

- `host`- The URL of ThingWorx Core to connect to (as a `STRING`)
- `port` - The number of the port to connect to (as an `INTEGER`)

Outputs

This service returns a `true` if the connection is successful, or `false` if it cannot connect.

Example

Here is an example of a REST call that tests a port:

```
http://localhost:8000/Thingworx/Things/LocalEms/Services/
    TestPort&host="myThingWorxServer.acme.com",port=443
```

UpdateConfiguration

The **UpdateConfiguration** service of the WS EMS allows you to change or replace its configuration file (`config.json`).

Tip

The changes or new configuration file that you pass in do not take effect until you restart the WS EMS. Use the Restart on page 73 service to force the changes to take effect.

Inputs

Pass in a `TW_INFOTABLE` that contains a JSON object. This object must contain

- `"config"` — A JSON string that is used to update or replace the current configuration file.
- `"replace"` — A Boolean that determines whether to update the current `config.json` file or to delete it entirely and replace it with the JSON string specified with the `"config"` parameter.

Outputs

This service returns an HTTP response only. If the operation was successful, it returns HTTP 200. Otherwise, it returns an HTTP error.

Example

Here is an example of a REST call that updates the configuration of a WS EMS running on your computer, using this service:

```
http://localhost:8000/Thingworx/Things/LocalEms/Services/
    UpdateConfiguration&"config":"config.json"/"replace":true
```

3

Working with the Lua Script Resource

This section provides information on how to work with the Lua Script Resource, which is used to implement things at the edge device level.

.

Getting Started with the Lua Script Resource

This section describes how to get started working with the Lua Script Resource by providing an overview of how to install the distribution.

From here, you need to do the following:

- Create a Lua Script Resource configuration file to set logging preferences, define Edge things that will run in the Lua Script Resource environment, define any extensions, etc. For more information, see Lua Script Resource Configuration File on page 80.
- Create a Lua Script Resource Template File to define properties, services, and tasks for Edge things. For more information, see Lua Script Resource Configuration on page 84.
- Run the Lua Script Resource. For more information, see Running the Lua Script Resource on page 91.

Installing the Lua Script Resource Distribution

The Lua Script Resource is included in the ThingWorx WS EMS distribution, which is available from PTC and is distributed as a simple .zip file.

Note

If the WS EMS and the Lua Script Resource are to exist on the same Edge device, you can skip this section as the Lua Script Resource was installed as part of Installing the ThingWorx WS EMS Distribution on page 16.

Install the WS EMS distribution on an edge device as follows:

1. Download the package that is correct for the operating system and platform that you plan to use:
   - MicroServer-*version*-Linux-ARM-HWFPU
   - MicroServer-*version*-Linux-ARM
   - MicroServer-version-Linux-coldfire
   - MicroServer-*version*-Linux-x86-32

- MicroServer-*version*-Linux-x86-64
- MicroServer-*version*-Win32

2. Following download, select a location for the WS EMS distribution on the file system of the edge device.
3. Unzip the WS EMS distribution.

WS EMS Distribution Contents

When unzipped, the ThingWorx WS EMS distribution creates the folder, `\MicroServer`, which contains the following files and subdirectories:

Item	Description
`wsems`	The WS EMS executable that is used to run the Edge MicroServer. **Note** Linux users must be granted permissions to this file. If you require FIPS support on the supported Windows platforms (win32), make sure you have the FIPS package, which contains the `ws-ems-fips.exe` file.
`luaScriptResource`	The Lua utility that is used to run Lua scripts, configure remote things, and integrate with the host system.. **Note** Linux users must be granted permissions to this file. If you require FIPS support on supported Windows platforms (win32), make sure you have the FIPS package, which contains the file,`luaScriptResource-fips.exe`, instead.
`\etc`	Directory that contains configuration files and directories for the luaScriptResource utility.
`\etc config.json.complete`	A reference file that contains all the configuration options available for the WS EMS. **Note** The `config.json.complete` file is intended to be used only as an example of the structure of a configuration file. It cannot be used as presented in your environment.
`\etc config.json.minimal`	A reference file that contains the basic settings that are required to establish a connection.
`\etc config.lua.example`	A reference file that contains a basic configuration for the luaScriptResource utility. A `config.lua` file is required to run the Lua engine.
`\etc\community`	Directory from which third-party Lua libraries are deployed. Examples of these libraries include the Lua socket library and the Lua XML parser, .
`\etc\custom`	Directory that will contain your custom scripts and templates.

Item	Description
`\etc\custom\templates`	Directory that contains an example `.lua` template and that is used to deploy custom templates.
`\etc\custom\scripts`	Directory from which custom integration scripts are deployed.
`\etc\thingworx`	Directory that contains ThingWorx-specific Lua files that are used by the Lua Script Resource (LSR). Do not modify this directory and its contents because an upgrade will overwrite any changes.
`\install_services`	Directory that contains the `install.bat` file for Windows bundles or the following files for Linux bundles: `install`, `tw_luaScriptResourced`, and `tw_microserverd`.

Configuration of the Lua Script Resource

This section describes how to configure the Lua Script Resource.

Lua Script Resource Configuration File

Named `config.lua`, the configuration file is a text file that is separated into sections, with a section that sets logging levels, another one that configures edge things to run in the scripting environment, and a final one that defines any Lua Script extensions that are to be used by the Lua Script Resource.

To view an example of a configuration file:

1. Open a command window or terminal session on the system or device that is hosting the Lua Script Resource.
2. Change into the `\MicroServer\etc` directory.
3. Open `config.lua.example`. Use this example as a reference while reading about the various sections of the configuration file.

Configuring the Log Level

The `log_level` section is used to configure the Lua Script Resource to collect logging information.

```
scripts.log_level = "INFO"
```

Logging Parameter

This logging section of the configuration file contains the following parameter:

Use	To
`log_level`	Specify the level of information included in the log file for the Lua Script Resource. Valid values, from least to most verbose, include: • FORCE • ERROR • WARN • INFO • DEBUG • TRACE

Configuring Edge Things

The `EdgeThing` section is used to configure an Edge thing to run in the Lua Script Resource environment. Of all the parameters that can be included in the `EdgeThing` section of the configuration file, only the `file` and `template` parameters are required. All other parameters are optional and can be included anywhere between the curly braces {}.

Note

The entry that defines the `thing.lua` Edge thing must exist in the configuration file. You can define additional Edge things with their own `EdgeThing` statements.

```
scripts.EdgeThing = {
    file = "thing.lua",
    template = "example",
}
```

Parameters

The `EdgeThing` section contains the following parameters:

Use	To
`file`	Define an Edge thing. This parameter is required.
`template`	Specify the template file associated with the Edge thing. The template file defines the behavior of the Edge thing. This parameter is required. The template file must be placed in the `\MicroServer\etc\custom\templates` folder.
`scanrate`	Specify how frequently to evaluate the properties and possibly push them to ThingWorx Core, in milliseconds. The default value is 60000

Use	To
	milliseconds.
`taskrate`	Specify how frequently to execute the tasks that are defined in the template of the edge thing, in milliseconds. The default value is 15000 milliseconds.
`scanRateResolution`	Specify how long the main execution thread for this edge thing pauses between iterations, in milliseconds. Each iteration checks the scan rate and task rate to determine if any properties are to be evaluated, or any tasks are to be executed. The value must be less than the scan rate or task rate. The default is 500 milliseconds. See also Configuring the scanRateResolution on page 82.
`register`	Specify whether or not the Edge thing registers with the WS EMS. The default value is true (recommended).
`keepAliveRate`	Specify how frequently this edge thing should renew its registration with the WS EMS, in milliseconds. If the WS EMS is restarted, this parameter controls the maximum amount of time before this edge thing is re-registered. This value also controls how frequently the WS EMS performs a keep-alive check on the Lua Script Resource. If the Lua Script Resource is unavailable, the registered thing is unbound from ThingWorx Core and appears to be offline. The default value is 60000 milliseconds.
`requestTimeout`	Specify the amount of time to wait for a response to an HTTP request to the WS EMS before timing out.
`maxConcurrentPropertyUpdates`	Specify the maximum number of properties that can be included in a single property update call to ThingWorx Core. This value can be decreased if the overall size of the batch property pushes is larger than what is supported by the WS EMS. The default value is 100 properties.
`getPropertySubscriptionsOnReconnect`	Specify whether or not the edge thing re-requests its property subscriptions when it reconnects to the WS EMS. This value is useful if the Lua Script Resource is running on a different edge device from the WS EMS. The default value is true.
`identifier`	Specify the identifier used to register the edge thing with the WS EMSand ThingWorx Core.
`useShapes`	Specify whether or not to use data shapes for property definitions. The default value is true.

Configuring the scanRateResolution

The `scanRateResolution` setting controls the frequency at which the main loop of a script for a thing executes in the LSR. Once it is started, a script enters into a loop that executes until the script resource is shut down. Each iteration of this main loop takes a number of actions, potentially increasing CPU usage.

At a high level, a script takes the following actions:

1. Goes through all your configured properties. If the `scanRate` amount of time for each property has expired, the property is evaluated to see if it should be pushed. To evaluate the property, the LSR calls the read method of the property handler for each property. The new value is compared to the old (if necessary). If it needs to be pushed to the server, the property and its value are added to a temporary list of properties to be pushed.
2. The temporary list of properties is pushed to the server. If no properties have been evaluated, or no properties need to be pushed, this list is empty and nothing is pushed.
3. The registration of the thing with the WS EMS is checked, using a call to the WS EMS.
4. If the `taskRate` time has expired, configured tasks are executed.
5. GC (garbage collection) is run if five seconds or more have elapsed.
6. The thread then sleeps for the number of milliseconds specified in the `scanRateResolution` parameter.

If you do not need the main loop to drive calls to the handlers that read your properties, you could set your `scanRateResolution` fairly high. A high setting would cause the main loop to sleep longer between iterations, which could have a few side effects:

1. The registration check will happen at the `scanRateResolution`, unless you adjust the `keepAliveRate` to be greater than the `scanRateResolution`.
2. You will need to set your `taskRate` and `scanRate` parameters to be greater than the `scanRateResolution`, or the script resource will complain during startup. Since it controls the pause of the main loop, the `scanRateResolution` is the main limiting factor in how often the main loop actions occur.
3. Shutting down the script resource can be delayed by up to the number of milliseconds specified for the `scanRateResolution` parameter, since the main loop must exit for the script to shut down.

The default value for the `scanRateResolution` is 500 milliseconds. If however, you do not require the loop to execute that often, consider setting this value much higher, even 10,000 milliseconds or more, to slow the execution of the loop and save CPU load.

Template Configuration for the Lua Script Resource

The template file is a text file that is separated into sections. Each section is used to define the overall behavior, properties, services, and tasks for edge things that reference the template file. Template files must be placed in the `\MicroServer\etc\custom\templates` folder.

To view an example of a template file:

1. Open a command window or terminal session on the system or device that is hosting the Lua Script Resource.
2. Change into the directory, `\MicroServer\etc\custom\templates`.
3. Open the file, `example.lua`. Use this example as a reference while reading about the various sections of the template file.

Including a Data Shape

The `require` statement pulls the functionality of a data shape into the template. A data shape can define properties, services, and tasks. If a template defines a property, service, or task that has the same name as one defined in the shape file, the definition in the shape file is ignored. Be careful that you do not have duplicate names for these characteristics.

```
require "yourshape"
```

The following table describes the `require` parameter:

Use	To
`require`	Specify the name of the shape file that contains the properties, services, and task definitions that are to be added to the template file configuration. The shape file must be located in the directory, `\MicroServer\etc\custom\shapes`. **Note** If you intend to use file transfer with this edge device, you must include the following statement: `require "thingworx.shapes.filetransfer"`

Configuring the Module Statement

The `module` statement is required. This statement tells the software component of the edge device to operate according to the configuration instructions contained in the template file that you specify.

```
module ("templates.example", thingworx.template.extend)
```

The following table describes the two parts of the `module` statement:

Part	Description
Name of template file	The first part of the `module` statement must contain the name of the template file that contains the configuration for the software component of the Edge device. For example, `template.acmedevice`.
Extension of the ThingWorx template	The second part of the module statement, `thingworx.template.extend`, identifies the file as a template file to the system and extends the base ThingWorx template implementation. Do not modify this part of the statement.

Configuring Data Shapes

The `dataShapes` section is used to define data shapes that describe the structure of an infotable.

Data shapes that are declared can be used as input or output for services. In addition, you can use data shapes to generate strongly typed data structures.

Each data shape is defined using a series of tables, with each table representing a field within the data shape. These fields must have a name and base type, but may also include other parameters.

```
dataShapes.MeterReading(
  { name = "Temp", baseType = "INTEGER" },
  { name = "Amps", baseType = "NUMBER" },
  { name = "Status", baseType = "STRING", aspects = {defaultValue="Unknown"} },
  { name = "Readout", baseType = "TEXT" },
  { name = "Location", baseType = "LOCATION" }
```

The following table lists and describes these parameters:

Use	To
`dataShape.nameOfDataShape`	Declare a data shape.
`name`	Required. Specify the name of a field in the data shape.
`baseType`	Required. Specify the ThingWorx base type of the field in the data shape. For a list of the ThingWorx base types, refer to ThingWorx Base Types.
`description`	Provides a description of the data shape.
`ordinal`	Specify the order.
`aspects`	Specify a table that can provide additional information, such as a default value, or a data shape if the base type of the field is `INFOTABLE`.

Defining Properties

The `properties` section is used to define the properties associated with an Edge thing. While remote data properties can be read on demand, you can also define data push rules so that the data does not have to be polled by ThingWorx Core from the edge device.

A property is defined by specifying a name, baseType, pushType, the threshold for determining a data change push to ThingWorx Core, and any other necessary parameters.

Note

When these properties are bound at ThingWorx Core, it respects the template settings. However, if changes to the push and cache settings are made at ThingWorx Core, those settings override the local template settings.

```
properties.IParametersnMemory_Imagelink =    { baseType="IMAGELINK", pushType="NEVER",
                                          value="http://www.thingworx.com" }
properties.InMemory_InfoTable =    { baseType="INFOTABLE", pushType="NEVER",
                                          dataShape="AllPropertyBaseTypes" }
properties.InMemory_Integer =      { baseType="INTEGER", pushType="NEVER", value=1 }
properties.InMemory_Json =         { baseType="JSON", pushType="NEVER", value="{}" }
properties.InMemory_Large_String = { baseType="STRING", pushType="NEVER",
                      value=string.rep("Lorem ipsum dolorsi ", 15000) .. "the end"  }
properties.InMemory_Location =     { baseType="LOCATION", pushType="NEVER",
       value = { latitude=40.03, longitude=-75.62, elevation=103 }, pushType="NEVER" }
```

The following table lists and describes the parameters for defining properties:

Use	To
`properties.nameOfProperty`	Declare a property.
`baseType`	Required. Specify the base type of the property.
`dataChangeType`	Provide a default value for the Data Change Type field of the property definition on ThingWorx Core, if the property is initially created using the Manage Bindings feature of ThingWorx Composer. Valid values include: • `ALWAYS` • `VALUE` • `ON` • `OFF` • `NEVER`
`dataChangeThreshold`	Provide a default value for the Data Change Threshold field of the property definition on ThingWorx Core, if the property is initially created using the Manage Bindings feature of ThingWorx Composer.
`pushType`	Specify whether the property should push new values upon change to ThingWorx Core. Valid values include:

Use	To
	• `ALWAYS` • `VALUE` • `NEVER` The default is `NEVER`. A `pushType`of `NEVER` does not push data to ThingWorx Core, so when a property with `pushType=NEVER` is queried on ThingWorx Core, it queries the software of the edge device for the data value. A `pushType` of `ALWAYS` pushes the data every time the property is read at the edge device, which is determined by the `scanRate` parameter. If the `scanRate` is not set on the property, the `scanRate` from the Lua Script Resource configuration file is used. If not defined in either location, a default of 60000 milliseconds (1 minute) is used. The Edge device pushes all properties that have a `pushType` of `ALWAYS` and the same scan rate in one call, rather than make individual calls per property. For `NUMBER`or `INTEGER`property types, a `pushType`of `VALUE`pushes data to ThingWorx Core only when the data value change exceeds the `DataChangeThreshold`setting.
`pushThreshold`	For properties with a `baseType` of `NUMBER`, and a `pushType` of `VALUE`. specify how much a property value must change before the new value is pushed to ThingWorx Core.
`handler`	Specify the name of the handler to use for property reads/writes. Valid values include: • `script` • `inmemory` • `http` • `https` • `generator` The default handler is `inmemory`. The `script`, `http`, and `https` handlers use the key field to determine the endpoint where their read/writes are to be executed. **Note** Custom handlers can specify other property attributes. When a handler is used to read or write a property, the entire property table is passed to the handler.
`key`	Define a key that the handler can use to look up or set the value of the property. In the case of a `script` handler, this key is a URL path. For `http` or `https` handlers, this key should be a

Use	To
	URL and not the protocol. This parameter is not required for `inmemory` or nil handlers.
`value`	Specify the default value of the property. The value is updated as the value changes on the Edge device. The default value is 0.
`time`	Specify the last time the value of the property was updated, in milliseconds since the beginning of the epoch. When things are created from this template, the current time is set automatically, unless a default value is provided in the definition of the property.
`quality`	Specify the quality of the value of the property. A default value should be provided for the quality. Otherwise, the value defaults to `GOOD` for properties without a handler, and `UNKNOWN` for properties with a handler.
`scanRate`	Specify the frequency of checking the property for a change event, in milliseconds. The default value is 5000 milliseconds (every 5 seconds). If you do not define a `scanRate`, the `scanRate` in the Lua Script Resource configuration file is used. Defining the `scanRate` within the property overrides the `scanRate` setting in the configuration file. See also Configuring the scanRateResolution on page 82.
`cacheTime`	Initialize cache time of value for property at ThingWorx Core. The default value is –`1`if the `dataChangeType`is `NEVER`, and is 0 when `dataChangeType`is `ALWAYS`or `VALUE`. If a value that is greater than 0 is specified, it is used by ThingWorx Core as the initial value for the cache time, and is applied only when using the browse functionality of ThingWorx Composer to bind the property.

Defining Services

The `serviceDefinition`section is used to provide metadata for a service that is used when browsing services from ThingWorx Core.

A service definition is a separate entry in the template file from the actual implementation of the service. The name of the service definition must match the name of the service it is defining in order for the service to work as expected.

```
serviceDefinitions.Add(
  input { name="p1", baseType="NUMBER", description="The first addend of the operation" },
  input { name="p2", baseType="NUMBER", description="The second addend of the operation" },
  output { baseType="NUMBER", description="The sum of the two parameters" },
  description { "Add two numbers" }
```

```
)

serviceDefinitions.Subtract(
  input { name="p1", baseType="NUMBER", description="The number to subtract from" },
  input { name="p2", baseType="NUMBER", description="The number to subtract from p1" },
  output { baseType="NUMBER", description="The difference of the two parameters" },
description { "Subtract one number from another" }
```

Parameters

The following table lists and describes the parameters that you can use to define services:

Use	To
`serviceDefinition.nameOfService`	Declare a Service Definition.
`input`	Describe an input parameter to the service that is referenced within the data table that is passed to the service at runtime. Valid values include: • `name` — The name of the input parameter. • `baseType` — The type of input parameter used by the service. • `description` — A description of the input parameter.
`output`	Describes the output produced by the service. Valid values include: • `baseType` — The type of output that the service produces. • `description` — A description of the output that the service produces.
`description`	Provide information about the purpose and operation of the service.

Implementing Services Using the Lua Script Engine

Use the `services` section to implement the services that you declared when defining services.

Once a service has been defined, you can implement custom logic for the service, using the Lua Script engine. To speed implementation, you can use the add-ons of the Lua community and ThingWorx-specific APIs. The APIs are included in the WS EMS distribution to facilitate the development of custom scripts.

Services are defined as Lua functions that can be executed remotely from ThingWorx Core, and must provide a valid response in their return statement. For example:

```
services.Add = function(me, headers, query, data)
  if not data.p1 or not data.p2 then
    return 400, "You must provide the parameters p1 and p2"
  end
  return 200, data.p1 + data.p2
end

services.Subtract = function(me, headers, query, data)
  if not data.p1 or not data.p2 then
    return 400, "You must provide the parameters p1 and p2"
  end
  return 200, data.p1 - data.p2
end
```

Parameters

The following table lists and describes the parameters that you can use to define a service:

Use	To
`services.nameOfService`	Implement a service. The name must match the name that is specified for the service in the service definition.
`me`	Create a table that refers to the thing.
`headers`	Create a table of HTTP headers.
`query`	Specify the query parameters from the HTTP request.
`data`	Create a Lua table that contains the parameters of the service call.

A service function must return the following values, in the following order:

1. An HTTP return code (200 for success).
2. A table of HTTP response headers that should contain a valid Content-Type header, typically with a value of application/json.
3. (Optional) A default table can easily be generated by calling **tw_utils.RESP_ HEADERS()**.
4. The response data, in the form of a JSON string. This data can be generated from a Lua table using `json.encode`, or **tw_utils.encodeData()**.

Configuring Tasks

Use the `task` section to define Lua functions that are executed periodically by the Lua Script Resource (LSR), such as background tasks, resource monitoring, event firing, and so on. These tasks allow you to introduce any customized functionality that you may need. You should follow the general pattern shown in the sample below:

```
tasks.Compare = function(me)
  -- Do task
end
```

The following table lists and describes the parameters that you can use to configure tasks:

Use	To
`task.nameOfTask`	Implement a task that is scheduled in the Lua Script Resource to run periodically.
`me`	Create a table that refers to the thing.
`end`	Define the end of the Lua function that defines the task.

Running the Lua Script Resource

The Lua Script Resource can be run either from a command line or as a service to host things on the remote device.

Running from a Command Line

To run the Lua Script Resource from a command line, follow these steps:

1. Open a command window or terminal session on the system or device that is hosting the Lua Script Resource.
2. Change to the `\MicroServer\etc` directory.
3. Copy and rename the `config.lua.example` file to `config.lua`.
4. Configure the file as necessary. See Lua Script Resource Configuration File on page 80.
5. Change directories back to the to the top level `\MicroServer` directory.
6. Enter the `luaScriptResource` command to run the Lua Script Resource executable. To include a specific configuration file, use a command similar to the following:

   ```
   luaScriptResource -cfg .\etc\config.lua
   ```

   Note

   If no configuration file is specified, the default file, `etc\config.lua`, is used.

   This command causes the Lua Script Resource to start listening on port 8001, if the default values are used.
7. To access the interface of the extension, open a browser and enter the following address to see a list of executing scripts:

   ```
   http://localhost:8001/scripts
   ```
8. Should you need to shut down the Lua Script Resource, press ENTER to display the console prompt and type `q`.

Running as a Service

To run the Lua Script Resource as a Windows service, follow these steps:

1. Open a command window or terminal session on the system or device that is hosting the Lua Script Resource.
2. Change to the `\MicroServer\etc` directory.
3. Copy and rename the `config.lua.example` file to `config.lua`.
4. Configure the file as necessary. See Lua Script Resource Configuration File on page 80.
5. Run the following command:

```
C:\Microserver\luaScriptResource.exe -cfg "C:\Microserver\etc\config.lua"
                                     -i "ThingWorx Script Resource"
```

Where:

`cfg`	Specifies the full path to the location of the configuration file.
`i`	Specifies the name to be used for the installed service.

Note

Run the `luaScriptResource` executable and the reference to the configuration file using the full path (even if it is running from the folder where the `luaScriptResource.exe` file is located).

Due to space constraints, the command shown above has the second argument/value pair on a second line. Do NOT just copy and paste this command without removing the extra line break and spaces.

6. Should you need to uninstall the service, run the following command:

```
C:\Microserver\luaScriptResource -u "ThingWorx Script Resource"
```

Where:

`u`	Specifies the name of the service to be un-installed. This name must exactly match the name that you assigned to the Lua Script Resource service.

Troubleshooting the Lua Script Resource

This section discusses issues that can arise when using the Lua Script Resource, along with recommended solutions.

Problem	Possible Solution
I have a tunnel configured on my thing (created using the **RemoteThing** thing template) on ThingWorx Core, but it is not working.	Verify that the Public Host and Public Port settings of the Tunnel Subsystem's Configuration are set to the externally available host name/IP and port.
The thing that I have configured in `config.lua` cannot communicate with my thing on ThingWorx Core.	Verify that the name of the thing in your `config.lua`matches the name of the thing on ThingWorx Core. You can also specify an identifier, if using matching names is a problem. See the "Configuring the WS EMS for File Transfer" section in WSEMS Example Configurations on page 47.

The WS EMS connects to ThingWorx Core, authenticates successfully, but the thing that I specified in the `auto_bind`section of my `config.lua`file is not being created on ThingWorx Core.	The `auto_bind` section is an array of objects. Verify that you enclosed the JSON object that represents your thing in square brackets as follows:
	`“auto_bind”: [{`
	`  “name”: “RemoteThing001”,`
	`  “gateway:” true`
	`}]`
	vs. the following, which leads to this scenario:
	`"auto_bind": { "name": "RemoteThing001", "gateway": true }`
	For more information about the auto_bind section and setting the gateway parameter, refer to Configuring Automatic Binding for WS EMS on page 36 and to Auto-bound Gateways on page 37

A

Base Types

Different operating systems and hardware have their own data types. For example an `int`could be 16 bits or 32 bits on different hardware. To standardize data types across hardware, ThingWorx Core and the ThingWorx Edge products use their own base types and *Primitives*. These types are portable. This section provides a table of the ThingWorx base types.

ThingWorx Base Types

The following table lists and briefly describes the ThingWorx Base Types:

ThingWorx Base Types

Type Name	Description
BOOLEAN	True or false values only
DATETIME	Date and time value
GROUPNAME	ThingWorx Group Name
HTML	HTML value
HYPERLINK	Hyperlink value
IMAGE	Image Value
IMAGELINK	Image link value
INFOTABLE	ThingWorx data structure used to define the results of a service or a set of properties for a thing
INTEGER	32–bit integer value
JSON	JSON structure
LOCATION	ThingWorx Location structure
MASHUPNAME	ThingWorx Mashup name
MENUNAME	ThingWorx Menu name
NOTHING	No type (used for services to define void result)
NUMBER	Double-precision value
STRING	String value
QUERY	ThingWorx Query structure
TEXT	Text value
THINGNAME	ThingWorx name for a thing
USERNAME	ThingWorx User name
XML	XML structure

B

Remote Things

This section explains what remote things are, briefly describes some of the templates available in ThingWorx Composer for remote things, and provides information about configuring them and their properties and services.

About Remote Things

A remote thing is a device or data source whose location is at a distance from the location of ThingWorx Core. A remote thing accesses ThingWorx Core, and can itself be accessed, over the network (Intranet/Internet/WAN/LAN). The device is represented on ThingWorx Core by a thing that is created using the **RemoteThing** template, or one of the derivatives of that template (for example, **RemoteThingWithFileTransfer**).

The default thing templates that you can use to create things for remote devices follow:

- **RemoteThing** — A basic thing that provides the ability to interact with a remote device.
- **RemoteThingWithFileTransfer** — A remote thing that can also transfer files.
- **RemoteThingWithTunnels** — A remote thing that supports tunneling.
- **RemoteThingWithTunnelsAndFileTransfer** — A remote thing that supports both file transfers and tunneling.
- **RemoteDatabase** — A remote OLE-DB data source
- **EMSGateway** — Addresses the WS EMS as a standalone thing. This template may be useful in situations where the WS EMS is running on a gateway computer and is handling communication for one or more remote things, which may reside on different IP addresses within a local area network
- **SDKGateway** — Similar to the **EMSGateway** template, but used when you are using an SDK implementation as a gateway.

Use the **RemoteThing**, **RemoteThingWithFileTransfer**, **RemoteThingWithTunnels**, and **RemoteThingWithTunnelsAndFileTransfer** templates for vendor-specific devices. The recommended approach is to use one of these templates to create a thing for your vendor-specific device, and when you have added the properties, services, and events for the thing, save it as your own template. That way, you can add more of the same devices quickly and easily by using your template as the base.

Configuring Remote Things

You can configure local properties and services for a remote thing. *Local properties* are properties similar to any other entity in the system. They are defined as local, they can be set through standard interfaces, they can be bound to other properties, but they are not directly bound to a data point from a remote device or data store. Local Services are services that execute on the server - although they may interact with data from remote data sources.

A *remote property* is a property that is directly connected to a remote data point or data object. It is read from the remote data, depending on the cache and push rules you define. Please refer to the section, Configuring Properties for Remote Things on page 99, for more details. Remote services are similar to remote properties.

A *remote service* is a reference to business logic running on the edge. When you call a remote service on ThingWorx Core, it relays that request to the bound edge service, and returns the result from the edge business logic. Please see the section, Configuring Services for Remote Things on page 99, for more information.

Configuring Properties for Remote Things

It is possible to browse the configuration of a remote thing and "mass" bind its remote properties. The ability to bind the properties of a remote device all at once allows you to fully configure the remote thing on the server, alleviating much of the manual configuration.

1. In ThingWorx Composer, on the **Properties** tab of the remote thing, click **Manage Bindings.**
2. In the **Manage Property Bindings** dialog box, click the **Remote** tab.
3. You will see a list of the properties of the remote device on the left. You can drag them individually (to the **Drag HERE to create new properties area**) or click **Add All.**
4. You can then individually edit the property details to suit your needs. You can also bind the remote properties to properties that you have already defined on ThingWorx Core. To bind remote properties to existing properties on ThingWorx Core, drag a remote property onto an existing thing property. When you bind properties from the edge, the cache and push settings from the edge are set based on the configuration settings of the edge device. You can choose to override them by changing the settings at ThingWorx Core.

   You can also manually create the properties by using the standard property configuration dialog box.

Configuring Services for Remote Things

There are two types of services for remote things, as follows:

- **Local (JavaScript)** - JavaScript business logic executing on the server.
- **Remote** - a direct call to a remote service on a remote thing (such as a custom-defined Lua script).

Remote Services and Binding

When you define a remote service, you are defining the metadata for the service so that it can be properly consumed from the server. The definition includes the service name at the server, the description, remote service name, and the inputs and outputs of the service. This will bind the service to the remote service, and the remote service is executed when the service runs. From a Mashup or REST API perspective, it will appear the same as a local service.

When the WSEMS opens a connection to ThingWorx Core, it goes
through a three step process:

1. Initiation: This establishes the physical WebSocket connection and prepares it to handle inbound and outbound messages.
2. Authentication: The WS EMS can authenticate using an application key. All communication that is passed over the connection from the WS EMS to ThingWorx Core will run under the security context of this application key. After authentication is complete applications can use the REST interface of WS EMS to interact with ThingWorx Core.
3. Binding: After authentication, remote things on ThingWorx Core can bind to the connection of the WS EMS. Binding is the process that notifies ThingWorx Core that a particular remote thing is associated with an established connection. After a thing is bound to a connection, ThingWorx Core will indicate a change in the value of its **isConnected** property to true and update its **lastConnection** property. ThingWorx Core can send outbound requests to thing.

You can also directly browse and bind to remote services if the remote thing is running and is connected. Click **Browse Remote Services** to view the services that are currenlty defined for the remote thing. You can then add them to the local service definitions through the drag and drop interface (similar to how you bind remote data properties).

The process of registering a thing with the WS EMS also causes the thing to bind. The de-registering of a thing causes it to unbind.