Guaranteed Delivery Developer's Guide

Version 10.11

October 2021

WEBMETHODS

Document ID: IS-GD-DG-1011-20211015

Table of Contents

About this Guide **5**
Document Conventions 6
Online Information and Support 7
Data Protection 7

1 Overview of Guaranteed Delivery **9**
Overview 10
What Is Guaranteed Delivery? 10
Indicating You Want to Use Guaranteed Delivery 10
How Transactions Are Managed 10
Identifying Transactions 12
Specifying How Long Transactions Are Active 12
Handling Failures 13

2 Creating a Java Client that Uses Guaranteed Delivery **15**
Overview 16
Sample Code (Synchronous Request) 17
Sample Code (Asynchronous Request) 20

3 Creating a Flow Service that Uses Guaranteed Delivery **23**
Overview 24
Sample Flow (Synchronous Request) 24
Sample Flow (Asynchronous Request) 25

About this Guide

- Document Conventions 6
- Online Information and Support 7
- Data Protection 7

Guaranteed delivery is a facility of webMethods Integration Server that ensures guaranteed, one-time execution of services and protects transactional requests from certain failures that might occur on the network, in the client, or on the server. This guide is for users who want to invoke services using guaranteed delivery from either a client application or another service.

Note:
This guide describes features and functionality that may or may not be available with your licensed version of webMethods Integration Server. For information about the licensed components for your installation, see the **Server > Licensing** page in webMethods Integration Server Administrator.

Document Conventions

Convention	Description
Bold	Identifies elements on a screen.
Narrowfont	Identifies service names and locations in the format *folder.subfolder.service*, APIs, Java classes, methods, properties.
Italic	Identifies: Variables for which you must supply values specific to your own situation or environment. New terms the first time they occur in the text. References to other documentation sources.
`Monospace font`	Identifies: Text you must type in. Messages displayed by the system. Program code.
{ }	Indicates a set of choices from which you must choose one. Type only the information inside the curly braces. Do not type the { } symbols.
|	Separates two mutually exclusive choices in a syntax line. Type one of these choices. Do not type the | symbol.
[ ]	Indicates one or more options. Type only the information inside the square brackets. Do not type the [ ] symbols.
...	Indicates that you can type multiple options of the same type. Type only the information. Do not type the ellipsis (...).

Online Information and Support

Software AG Documentation Website

You can find documentation on the Software AG Documentation website at https://documentation.softwareag.com.

Software AG Empower Product Support Website

If you do not yet have an account for Empower, send an email to empower@softwareag.com with your name, company, and company email address and request an account.

Once you have an account, you can open Support Incidents online via the eService section of Empower at https://empower.softwareag.com/.

You can find product information on the Software AG Empower Product Support website at https://empower.softwareag.com.

To submit feature/enhancement requests, get information about product availability, and download products, go to Products.

To get information about fixes and to read early warnings, technical papers, and knowledge base articles, go to the Knowledge Center.

If you have any questions, you can find a local or toll-free number for your country in our Global Support Contact Directory at https://empower.softwareag.com/public_directory.aspx and give us a call.

Software AG Tech Community

You can find documentation and other technical information on the Software AG Tech Community website at https://techcommunity.softwareag.com. You can:

- Access product documentation, if you have Tech Community credentials. If you do not, you will need to register and specify "Documentation" as an area of interest.
- Access articles, code samples, demos, and tutorials.
- Use the online discussion forums, moderated by Software AG professionals, to ask questions, discuss best practices, and learn how other customers are using Software AG technology.
- Link to external websites that discuss open standards and web technology.

Data Protection

Software AG products provide functionality with respect to processing of personal data according to the EU General Data Protection Regulation (GDPR). Where applicable, appropriate steps are documented in the respective administration documentation.

1 Overview of Guaranteed Delivery

- Overview 10
- What Is Guaranteed Delivery? 10
- Indicating You Want to Use Guaranteed Delivery 10
- How Transactions Are Managed 10
- Identifying Transactions 12
- Specifying How Long Transactions Are Active 12
- Handling Failures 13

Overview

This chapter explains what guaranteed delivery is, how to indicate that you want to use guaranteed delivery services from a client application (an Integration Server or standalone Java program) or from another service, how to customize Job Managers to manage guaranteed delivery transactions, what determines how long transactions remain active, and how errors are handled.

Note:
This guide describes how to invoke services using guaranteed delivery from either a client application or another service. For more information about guaranteed delivery, including how to configure the webMethods Integration Server for guaranteed delivery and how to shut down and initialize guaranteed delivery transactions, see *webMethods Integration Server Administrator's Guide*.

What Is Guaranteed Delivery?

Guaranteed delivery is a facility of webMethods Integration Server that ensures guaranteed, one-time execution of services. It protects transactional requests from *transient* failures that might occur on the network, in the client, or on the server.

A transient failure is a failure that can correct itself within a specified period of time. If a request cannot be delivered to the server due to a transient failure, the request is resubmitted. If the problem corrected itself, the request is successfully delivered on a subsequent attempt. You can determine what constitutes a transient error by specifying a time-to-live (TTL) period for a guaranteed delivery transaction and, optionally, the number of times a transaction should be retried. If you do not specify the TTL or retry value, the configured defaults are used.

You can use guaranteed delivery when you invoke a service from a client or from within another service.

Important:
You can only use the guaranteed delivery capabilities with stateless (that is, atomic) transactions. As a result, guaranteed delivery capabilities cannot be used with multi-request conversational services.

Indicating You Want to Use Guaranteed Delivery

To invoke services using guaranteed delivery from either a client application or another service use the class watt.client.TContext (TContext) that is part of the Client API. Similar to the standard class watt.client.Context (Context), you use TContext to request that webMethods Integration Server execute a service. However, the server performs guaranteed delivery functions when a client application or service requests services through TContext.

How Transactions Are Managed

Guaranteed delivery transactions are managed by Job Managers. For client applications, the Job Manager runs on the client. For services, the Job Manager runs on the server.

The Job Managers manage all guaranteed delivery transactions that a process creates using TContext. The Job Managers maintain a job store of the guaranteed delivery transactions. The job store contains a record for each transaction. In addition, the Job Managers maintain a log that tracks the progress of all transaction operations.

The Job Manager handles the invocation of the service using background threads, which the Job Manager allocates from a configurable pool of threads. The Job Manager sends the service requests to a webMethods Integration Server and accepts the results on behalf of the client applications or services that use TContext. If the Job Manager does not receive a result for a transaction in its job store, it resubmits that request to execute the service. It continues to resubmit requests until it either receives a result or the transaction expires.

Note:
For client applications, a single Job Manager runs in the client process and is shared by multiple TContext instances. For services, a single Job Manager runs in the server process and is shared by all TContext instances.

Customizing the Job Manager

You can customize how the Job Manager manages guaranteed delivery transactions programmatically or through system properties. To specify programmatically, your client application must specify the setting with the parameters of TContext methods. To specify through system parameters, specify the setting on the Java command line.

If a setting is specified both with a parameter of TContext and through a system property, the Job Manager uses the setting specified through the system property.

- **Location of the client transaction log.** Specify the file in which the Job Manager maintains its log of all the guaranteed delivery transaction operations for clients that are standalone Java programs.

  Tcontext Method: Specify using a parameter with the init method.

  System Property: Use the `-Dwatt.tx.logfile =` *filename* option. If a parameter is supplied to the TContext.init method and watt.tx.logfile is set, the value in watt.tx.logfile is used. If neither is set, the default is .\tx.log.

- **Submission interval for the Job Store.** Specify the number of seconds between sweeps of the job store. The Job Manager sweeps the job store to submit transactions to a webMethods Integration Server.

  TContext Method: Cannot specify using a TContext method.

  System Property: Use the `-Dwatt.tx.sweepTime=` *seconds* option.

  The default is: 60 seconds

- **Time to Retry Interval.** Specify the number of seconds to wait after a service request failure before the Job Manager resubmits the request to webMethods Integration Server.

TContext Method:	Cannot specify using a TContext method.
System Property:	Use the `-Dwatt.tx.retryBackoffTime=` *seconds* option.

  The default is: 60 seconds

- **Number of Client Threads in Thread Pool.** Specify the number of threads you want to make available in a thread pool to service pending requests.

TContext Method:	Cannot specify using a TContext method.
System Property:	Use the `-Dwatt.tx.jobThreads`.

  The default is: 5 threads

Identifying Transactions

It is the responsibility of the client application or service to obtain a *transaction ID* (tid) for each guaranteed delivery request and to specify the transaction ID with each subsequent request for the transaction.

The client application or service obtains the transaction ID from webMethods Integration Server using the startTx( ) method, which is used to start a guaranteed delivery transaction. See "Creating a Java Client that Uses Guaranteed Delivery" on page 15 and "Creating a Flow Service that Uses Guaranteed Delivery" on page 23 for additional instructions and sample code.

Specifying How Long Transactions Are Active

A guaranteed delivery transaction has two attributes that determine how long it stays active: the time-to-live (TTL) and the retry limit. The TTL specifies the number of minutes that a transaction is to remain active. The retry limit specifies the maximum number of times that the Job Manager is to resubmit a request. A transaction becomes inactive when the TTL or the retry limit (if specified) is reached, whichever comes first. When a transaction becomes inactive, it remains in the job store, but the Job Manager no longer attempts to submit the request.

The client application or service sets the TTL (and optionally, the retry limit) with the startTx () method, which it uses to start a guaranteed delivery transaction. See "Creating a Java Client that Uses Guaranteed Delivery" on page 15 and "Creating a Flow Service that Uses Guaranteed Delivery" on page 23 for additional instructions and sample code.

These values determine the degree of tolerance the client application or service has towards transient network and server errors that occur at run time. Specifically, they determine the length of the outage that the client application or service considers transient. An outage that exceeds these limits will be deemed unrecoverable by the Job Manager and will cause the Job Manager to return an error for the request.

Handling Failures

If a non-transient error prevents your client application or service from receiving the results from a service request, your application will receive an error message.

Records remain in the job store for a transaction until the client application or service explicitly ends the transaction. To avoid exhausting the job store, a client application or service must make sure to complete all the transactions it starts, or a site must establish administrative procedures to address failed jobs.

TContext can return the following types of errors:

- **AccessException.** The client application or service either supplied invalid credentials or is denied access to the requested service.
- **ServiceException.** The service encountered an execution error.
- **DeliveryException.** The Job Manager failed and became disabled. An administrator should be notified to correct this problem. For client applications, code your client application to notify an administrator when this type of error occurs. After the problem is corrected, re-enable the Job Manager using the TContext.resetJobMgr( ) method.

  For services, guaranteed delivery notifies the administrator identified by the watt.server.txMail configuration setting. After the problem is corrected, re-enable the Job Manager by executing the pub.tx:resetOutbound service.
- **IllegalRequestException.** The client application or service made an invalid request; for example, supplied an invalid transaction ID (tid) or other invalid parameter.
- **TXException.** A failure occurred with the transaction. The transaction timed out, hit the retry limit, or encountered a heuristic error. Typically, this type of error indicates that the transaction became inactive either because the time-to-live (TTL) value elapsed or the retry limit was met. To distinguish between these two errors, use the isExceededRetries( ) method.

  Heuristic errors will only occur if you altered the default configuration of webMethods Integration Server to fail PENDING requests when webMethods Integration Server is restarted after a failure. Use the isHeuristicFailure( ) method to determine if a heuristic error occurred.

  Note:
  A heuristic error does not guarantee that your transaction was not executed, only that its results could not be returned. Keep this in mind if you are processing transactions that must be executed once and only once (for example, an application that enters purchase orders or pays invoices). You might also need to implement additional mechanisms in your client application or service to ensure that a transaction does not get posted twice.

2 Creating a Java Client that Uses Guaranteed Delivery

- Overview 16
- Sample Code (Synchronous Request) 17
- Sample Code (Asynchronous Request) 20

Overview

Using the TContext function, you can submit requests from a Java client application that uses guaranteed delivery.

Creating a Java client that uses guaranteed delivery involves the following general steps:

1. Make sure the following are in your classpath:

   Integration Server_directory \lib\wm-isserver.jar

   (or *Integration Server_directory* \instances*instance_name*\lib\wm-isserver.jar)

   Software AG_directory \common\lib\wm-isclient.jar

   Software AG_directory \common\lib\ext\mail.jar

   Note:
   These jar files must be the same version as those present on the Integration Server to which your client program connects. If you are creating a stand-alone client application, you can obtain a copy of the jar files from the Integration Server. If you are creating a Java service for an Integration Server, verify that the Integration Server on which you deploy the service and the Integration Server to which the service submits guaranteed-delivery requests are both running the same version of Integration Server software.

2. Initialize TContext when a process starts. The server handles this function when a service uses guaranteed delivery.

3. Create TContext instances for different connection attributes. If you are only connecting to one host with a single set of credentials, you need only one TContext regardless of how many threads share the TContext.

   The main difference between Context (the standard class) and TContext is that your client application or service is responsible for obtaining a transaction ID (tid) and associating it with each request you make for the same transaction. You receive a transaction ID (tid) when you start a guaranteed delivery transaction.

4. After a transaction is started and a transaction ID is received, invoke a service using guaranteed delivery. You must supply the transaction ID when you invoke the service.

5. When the transaction completes, end the transaction to clear the record for the transaction from the Job Manager's job store.

   You can chain transactions in a sequence so that each transaction in a sequence waits until the preceding transaction executes. To chain transactions, supply the transaction ID (tid) from the previous transaction when starting a new transaction.

6. When you are finished executing guaranteed delivery transactions for a specific instance, disconnect to end the instance of TContext. When you disconnect, TContext unregisters the instance with the Job Manager.

   After a client application disconnects all TContext instances, it should shut down guaranteed delivery for the process. The server handles this function automatically when a service uses

guaranteed delivery. If your client application or service has active TContext instances when the shutdown occurs, the server throws an exception (unless the shutdown was performed with the force option).

The following examples show how you would submit both synchronous and asynchronous requests from a Java client to the Job Manager.

Sample Code (Synchronous Request)

The following code fragment illustrates the basic steps required to submit a synchronous request to the Job Manager. Synchronous requests are submitted using the invokeTx method. You can also submit asynchronous requests to the Job Manager as shown in the next section.

Important:
To compile the following sample code (or any Java client that uses guaranteed delivery), you must include the following import statements in your Java program.

```
import com.wm.data.*;
import com.wm.app.b2b.client.*;
import com.wm.util.*;
import com.wm.app.b2b.client.lic.*;
```

```
1   TContext tc = null;
    ClientKeyInfo.setGuaranteedDeliveryLicensed(true);
    // initialize TContext and establish connection attributes
    try {
2     TContext.init("./jobs", "./tx.log");
3     tc = new TContext();
4     tc.connect("localhost:5555", Administrator", "manage");
    } catch (ServiceException e) {
      System.err.println("Error: "+e.getMessage());
      System.exit(-1);
    }

    // do work with TContext - get tid, call service, end tid
    try {
5     String tid = tc.startTx(3);
6     IData result = tc.invokeTx(tid, "wm.server", "ping", IDataFactory.create());
      System.out.println("Result="+result.toString());
7     tc.endTx(tid);
    } catch (TXException e) {
      System.err.println("Job Failed: "+e.getMessage());
      System.exit(-1);
    } catch (DeliveryException e) {
      System.err.println("JobMgr Disabled: "+e.getMessage());
      System.exit(-1);
    } catch (AccessException e) {
8     System.err.println("Access Denied: "+e.getMessage());
      System.exit(-1);
    } catch (ServiceException e) {
      System.err.println("Error: "+e.getMessage());
      System.exit(-1);
    }

    // release connection and shutdown
    try {
      tc.disconnect();
9     TContext.shutdown();
10  } catch (ServiceException e) {
      System.err.println("Error: "+e.getMessage());
      System.exit(-1);
    }
```

#	Step	Description
1	Declare TContext.	Declare TContext as a variable.
2	Initialize TContext.	Initialize TContext and specify the job store directory and audit-trail log. The Job Manager starts. **Important:** Do not include this step if your client will run as a service on a webMethods Integration Server. This function is automatically performed by the server and must not be included in your code.
3	Instantiate TContext.	Create a new TContext object.

#	Step	Description
4	Establish connection attributes for the TContext instance.	Execute connect() to specify the webMethods Integration Server on which you want to invoke services using this context. You must connect as a user who is a member of the Administrators group on the Integration Server. **Note:** Multiple threads can share an instance of TContext as long as they use the same connection attributes—i.e., they use the same webMethods Integration Server and user ID/password (i.e., Administrator/manage) established by that instance of TContext. To set other connection attributes, use methods in the class Context such as the protocol to use (HTTP or HTTPS) and the proxy to use.
5	Start the transaction.	Execute startTx() to obtain a transaction ID (tid) and specify the transaction time-to-live (TTL).
6	Invoke the service.	Execute invokeTX() to invoke a service. Note that you pass the transaction ID (tid) as the first parameter to this method.
7	End the transaction.	Execute endTx() to end the transaction. This method clears the record for this transaction from the Job Manager's job store.
8	Check for errors.	Check for the different types of errors. Always check for Service Exceptions last.
9	Close the session on webMethods Integration Server.	Execute disconnect to end the use of this instance of TContext. The application should not perform this step until it is done because disconnect unregisters TContext with the Job Manager.
10	Shutdown.	The Job Manager ends. **Important:** Do not include this step if your client will run as a service on a webMethods Integration Server. This function is automatically performed by the server and must not be included in your code.

For additional information about TContext and its methods, see the TContext class in the *webMethods Integration Server Java API Reference*.

Sample Code (Asynchronous Request)

The following example illustrates the steps you take to submit an asynchronous request to the Job Manager. To submit an asynchronous request, you establish a connection and start a transaction just like you do for a synchronous request. However, you submit the request using the submitTX method instead of the invokeTx method. Then, you must retrieve the results of the request using the retrieveIDTx method (to get results as an IData object).

For additional information about TContext and its methods, see the TContext class in the *webMethods Integration Server Java API Reference*.

```
/**
 * Sample of a Java TContext client that uses SSL to perform a GD transaction */
import com.wm.data.IDataFactory;
import com.wm.data.IData;

import com.wm.app.b2b.client.*;
import com.wm.util.*;
import com.wm.app.b2b.client.lic.*;
```

```
public class TCSample {

  public static void main (String[] args)
  {
   TContext tc = null;
   ClientKeyInfo.setGuaranteedDeliveryLicensed(true);
```

```
   String privkey = "./config/privKey1.der";
   String[] certFiles = {"./config/cert1.der","./config/cacert1.der"};
   // initialize TContext and establish connection attributes
   try {
    TContext.init("./jobs", "./tx.log");
    tc = new TContext();
    tc.connect("localhost:5555", "Administrator", "manage");
    tc.setSecure(true);
    tc.setSSLCertificates(privKey,certFiles);
   } catch (ServiceException e) {
    System.err.println("Error: "+e.getMessage());
    System.exit(-1);   }
   // do work with TContext - get tid, call service, end tid
   try {
    String tid = tc.startTx(3);

    // Make an asynch call to invoke the specified transaction.
    tc.submitTx(tid, "wm.server", "ping", IDataFactory.create());

    // Retrieve the results of an asynch call in blocking mode.
    IData result = tc.retrieveTx(tid);

    System.out.println("Result="+result.toString());
    tc.endTx(tid);
   } catch (TXException e) {
    System.err.println("Job Failed: "+e.getMessage());
    System.exit(-1);
   } catch (DeliveryException e) {
    System.err.println("JobMgr Disabled: "+e.getMessage());
    System.exit(-1);
```

```
   } catch (AccessException e) {
    System.err.println("Access Denied: "+e.getMessage());
    System.exit(-1);
   } catch (ServiceException e) {
    System.err.println("Error: "+e.getMessage());
    System.exit(-1);
   }
   // release connection and shutdown
   tc.disconnect();
   TContext.shutdown();
  }
}
```

3 Creating a Flow Service that Uses Guaranteed Delivery

- Overview 24
- Sample Flow (Synchronous Request) 24
- Sample Flow (Asynchronous Request) 25

Overview

Using the services in the pub.remote.gd folder, you can build flow services that submit requests to other webMethods Integration Servers through guaranteed delivery.

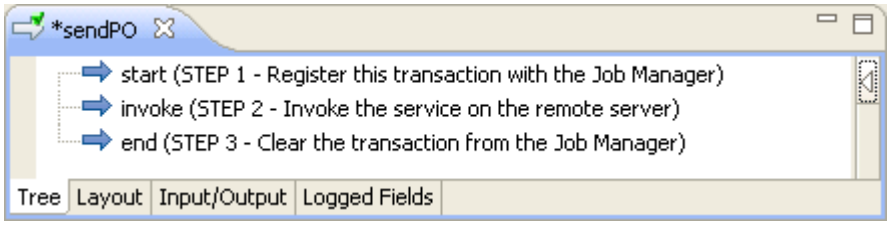
Note:
The Integration Servers that participate in a guaranteed-delivery transaction must both be running the same version of Integration Server software.

The following examples show how you would submit both synchronous and asynchronous requests using the built-in services. For a description of these services, see *webMethods Integration Server Built-In Services Reference*.

Sample Flow (Synchronous Request)

The following flow illustrates the basic steps you use to execute a synchronous transaction from a flow service.

Flow service that executes a synchronous transaction



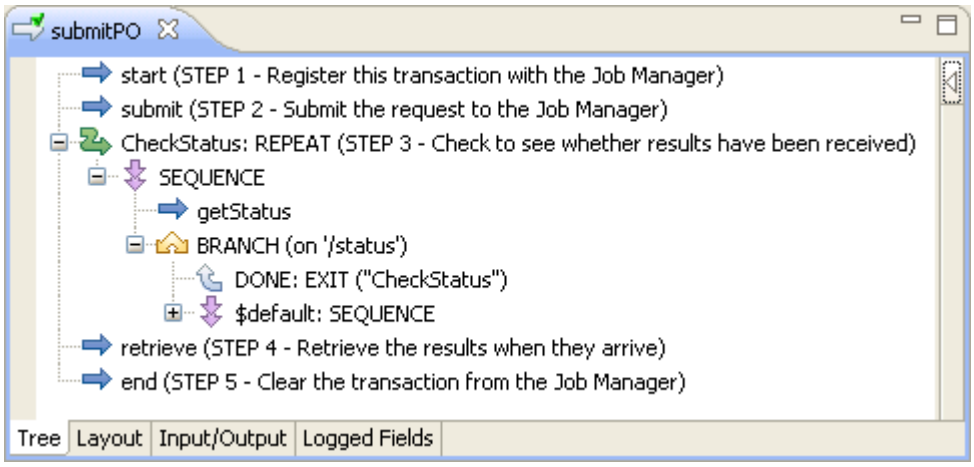
Step	Invoke this Service...	To...
1	pub.remote.gd:start	**Start the transaction.** When you invoke this service, you specify the alias for the webMethods Integration Servers to which you want to submit a request as well as transaction-related parameters such as time-to-live and followid. This service returns a tid as output. **Note:** Internally, this service opens a session on the server and performs startTx, so there is no need for you to explicitly open a session on the server like you must do in a Java guaranteed-delivery client.
2	pub.remote.gd:invoke	**Invoke the service.** You must provide the tid (produced by start, above), the name of the requested service, and the input values for that service as input. This service returns the results from the remote service as output.

Step	Invoke this Service...	To...
3	pub.remote.gd:end	**End the transaction.** You must call this service to clear the transaction from the job store. It takes the tid as input.

Sample Flow (Asynchronous Request)

The following flow illustrates the basic steps you use to execute an asynchronous transaction from a flow service.

Flow service that executes an asynchronous transaction



Step	Invoke this Service...	To...
1	pub.remote.gd:start	**Start the transaction.** When you invoke this service, you specify the alias for the webMethods Integration Server to which you want to submit a request as well as transaction-related parameters such as time-to-live and followid. This service returns a tid as output. **Note:** Internally, this service opens a session on the server and performs startTx, so there is no need for you to explicitly open a session on the server like you must do in a Java guaranteed-delivery client.
2	pub.remote.gd:submit	**Submit the service request.** You must provide the tid (produced by start, above), the name of the requested service, and the input values for that service as input.
3	pub.remote.gd:getStatus	**Check for results.** You can optionally use a REPEAT step to poll the job store and check whether the results

Step	Invoke this Service...	To...
		from the transaction have been received. This service returns “DONE” when results are available.
4	pub.remote.gd:retrieve	**Retrieve the results.** This service returns the results from the service request you submitted earlier. It takes the tid as input.
5	pub.remote.gd:end	**End the transaction.** You must call this service to clear the transaction from the job store. It takes the tid as input.