# TIBCO DataSynapse GridServer® Manager

## Service-Oriented Integration Tutorial

*Version 7.1.0*
*July 2022*

# Contents

# Introduction

This tutorial explains the basic features of the GridServer Services-oriented integration approach by means of several sample programs. The code for the tutorial programs is available from the GridServer SDK, which can be downloaded from the **Download Components** panel in the top navigation bar of the Administration Tool.

## Overview

GridServer hosts Services in an automatically scalable, load balanced, and fault-tolerant environment. Services can be written in a variety of languages and do not need to be compiled or linked with DataSynapse code.

Service implementations can be constructed using any of the following options:

- Arbitrary Java classes
- Arbitrary .NET classes
- Dynamic Libraries (.so, .dll) with methods that conform to a simple input-output string interface
- R functions
- Commands, such as a script or binary

These Services can be accessed in one of two ways:

- Service API — a client-side API in Java, COM, C++, R, or .NET
- Service Proxy — GridServer-generated C# or Java client stubs

The basic Service execution model is the same as that of other distributed programming solutions: method calls on the client are routed over the network, ultimately resulting in method calls on a remote machine, and return values make the reverse trip.

The chief benefit of hosting Services on GridServer over other approaches is that it virtualizes the Service. Rather than send a request directly to the remote machine hosting the Service, a client request is sent to the GridServer Manager, which enqueues it until an

Engine is available. The first Engine to dequeue the request hosts the Service. Subsequent requests might be routed to the same Engine or might result in a second Engine running the Service concurrently. This decision is based on the Service's priority, the amount of resources the Service has received in comparison with other Services, and how much state related to the Service resides on the first Engine. If an Engine hosting a Service fails, another takes its place. This mechanism, in which a single virtual Service instance (the client-side object) is implemented by one or more physical instances (Engine processes) provides for fault tolerance and essentially unlimited scalability.

# The Integration Process

Using the Service-Oriented integration approach involves six steps:

1. **Writing the Service, or adapting existing code**. A Service can be virtually any type of implementation: a library (DLL or .so), a .NET assembly, a Java class, an R function, even an executable. No DataSynapse libraries need be linked, but the remotely callable methods of the Service might have to follow certain conventions. These conventions are described later.
2. **Deploying the Service**. The implementation files and other resources required by a Service must be accessible from all Engines. This can be accomplished with a shared file system or GridServer's file update mechanism.
3. **Registering the Service Type**. To make the Service Type visible to clients, it must be registered, using the Administration Tool.
4. **Creating a Service Session**. The client creates or gets access to a Service before using it — no discovery or binding is required. Each Service Session might have its own state. Because of virtualization, a single Service Session can correspond to more than one physical instance of the Service, such as more than one Engine running the Service's code. Multiple asynchronous calls to a Service usually result in more than one Engine creating and accessing those Services.
5. **Making requests**. The methods or functions of a Service can be called remotely either synchronously or asynchronously.
6. **Destroying the instance**. Clients must destroy a Service Session when they are done with it, to free resources.

# Argument Types

Almost any public Java or .NET class with a public, no-argument constructor can be made into a Service with little or no modification. Each public method of the class can be accessed remotely through a Service operation. The input and output arguments must be either Serializable objects, support setter-getter interfaces in Java (beans) or public data members in .NET, or be pure strings — accessible as `String` in Java, `string` in .NET, `std::string` in C++, or `(char*, int)` in C. A dynamic library can be made into a Service if the functions exported as Service operations follow a simple string interface convention. The characteristics of each argument calling type are:

- **Serializable** — Services can use rich objects with serialization being accomplished through .NET and Java-specific data protocols. This is the simplest and most efficient way of exchanging data, but disallows interoperability between languages; for example, Java clients can only use Java Services and .NET clients can only use .NET Services.
- **XML Serializable** — .NET and Java objects can be written in such a way that they can be used with each other by converting them to XML.
- **Strings** — This approach allows for maximum interoperability among clients and Services implemented in different languages. For example, .NET clients can talk to Java or C++/C Services, and so on. Also, automatic string conversion allows Java and .NET Services with non-string arguments to be called using String arguments.

# A Simple Service in Java

This section describes how to create and access a simple Service in Java.

## The Service Implementation

In this section, let us create a simple Service in Java. The implementation's class is called `JavaAdder` and has one method, `add`, that takes two `doubles`. The method returns a `double` that is the sum of its arguments.

```
package examples.adder.service.JavaAdder;
public class JavaAdder {
    public double add(double a, double b) {
        return a + b;
    }
}
```

## Deployment and Grid Libraries

Once the class is written and compiled, it must be deployed as a Grid Library.

*Grid Libraries* are the method of deploying resources to Engines. They are an archive containing a set of resources and properties necessary to run a Grid Service, along with configuration information that describes how those resources are to be used.

To create and deploy a Grid Library:

1. Create a JAR file containing the necessary class file (`JavaAdder.class`).
2. Create a `grid-library.xml` file, like the one below:

```
<?xml version="1.0" encoding="UTF-8"?>
<grid-library>
    <grid-library-name>adder</grid-library-name>
    <grid-library-version>1.0.0.1</grid-library-version>
    <jar-path>
```

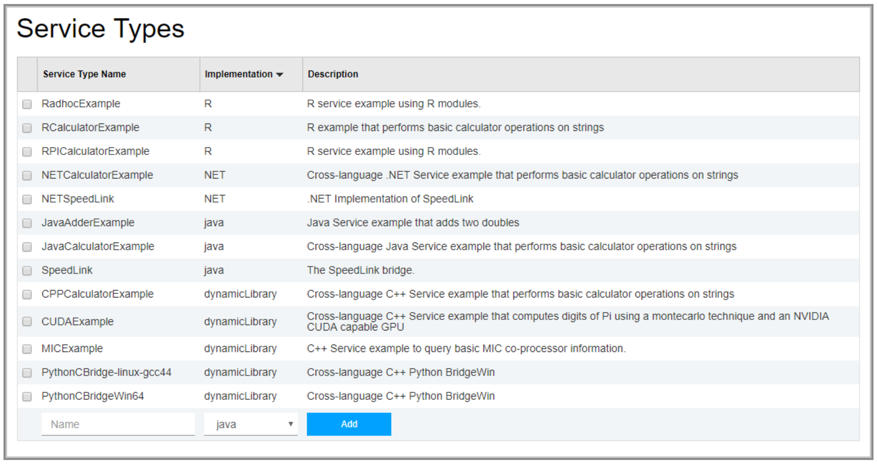
```
        <pathelement>jars</pathelement>
    </jar-path>
</grid-library>
```

3. Create an `adder` directory in a temporary location, and place the `grid-library.xml` file in it.
4. Create a `jars` directory within the `adder` directory, and place the JAR file containing the class in it.
5. Zip all contents under `adder` directory, making sure the directory `adder` is not included in the ZIP archive, to create an `adder-1.0.0.1.zip` Grid Library. If you're using UNIX, you can also create a `tar.gz` file of the directory instead (such as `adder-1.0.0.1.tar.gz` or `adder-1.0.0.1.tgz`).
6. Log in to the GridServer Administration Tool and go to **Services > Services > Grid Libraries**.
7. At the top right corner of the page, click **Upload Grid Library**, browse to and select your Grid Library file, then click **Upload**.
8. Select your Grid Library in the list and click **Deploy**.

More information about Grid Libraries and the deployment process is described the *TIBCO GridServer® Administration* and the *TIBCO GridServer® Developer's Guide*.

# Registering a Service Type

Service Types must be registered from the Administration Tool, at **Services > Services > Service Types**.




A list of existing Service Types appears on that page, along with a line for adding a new Service Type. Enter the Service Type name on the blank line. Let us name the Service Type `JavaAdderExample`. Now select Java as the Service implementation, then click **Add**.

In the window that appears after clicking the **Add** button, enter the fully qualified class name for the Service Type, which in this case is `examples.adder.service.JavaAdder`. The window also allows you to enter options for the Service Type. Enter a `*` in the **serviceMethods** field, indicating that all public methods might be called as Service Type methods.

> **Note**
> You must set the Grid Library's name in the Service Type's **gridLibrary** field.

# The Client Application

Having deployed and registered the Service Type, we are now ready to use it. There are two different techniques we can use to access this Service from a client:

- The Service API
- GridServer-generated proxy

Let us first demonstrate accessing the Service using the Service API in Java. Refer to the Javadocs on the **Documentation** panel in the top navigation bar of the Administration Tool. The `Service` class is used to access the Service either synchronously or asynchronously. This section consists largely of small code snippets. All of the code can be found in a single class, `example.adder.client.AdderClient`.

Only a few lines of code are needed to use the `JavaAdder` Service:

```
// Create a Service instance of JavaAdderExample
Service s = ServiceFactory.getInstance().createService
("JavaAdderExample");
// Perform Synchronous add
Object[] arguments = new Object[] { new Double(5), new Double(6) };
Double sum = (Double)s.execute("add", arguments);
System.out.println("Result of add: " + sum);
```

The first line gets an instance of the `ServiceFactory` class and calls its `createService` method to create a `Service` instance for the Service. If you try to create a Service Type whose name was not registered, `createService` throws a `GridServerException`.

The second line prepares the arguments to be submitted to the Service, two `Doubles` (note the use of the primitive wrapper object). The third line executes the `add` method with the given argument list and blocks until the method completes and its return value makes its way back to the client. If the remote method throws an exception, or an error occurs in the course of processing the request, an exception is thrown. `GridServerException` might wrap another exception; use its `getCause` method to obtain the wrapped exception.

A Java client does not terminate automatically, and pends after submitting Services and collecting results. Instead of calling `System.exit()` to terminate the client, you can set a property to change this behavior. Add this to your code:

```
DriverManager.setProperty(DriverManager.IS_DAEMON,
                          Boolean.TRUE.toString());
```

When this property is set to true, all client threads are daemon threads, meaning that the allow the process to shut down when all threads have shut down.

# Asynchronous and Parallel Processing Requests

A client can use the `submit` method instead of the `execute` method of `Service` to perform a remote invocation without waiting for the result. This allows the caller to make many asynchronous requests in parallel — with these requests being executed in parallel if the resources are available to do so. In addition to the method name and argument, the `submit` method takes a callback object that implements the `ServiceInvocationHandler` interface. This interface has a `handleResponse` method that is called with the method's return value in the event of a normal response, and a `handleError` message that is called if an error occurs. The submit method returns an integer that uniquely identifies the particular call; this unique ID is passed as the second argument to the `ServiceInvocationHandler` methods, so that you can match the response with the request if need be.

To use the submit method, we first require a class implementing `ServiceInvocationHandler`. The `handleResponse` method displays the response and adds it to a total.

```
// Handler for Service Clients
static class AdderHandler implements ServiceInvocationHandler {
    public void handleError(ServiceInvocationException e, int id) {
        System.out.println("Error from " + id + ": " + e);
    }
    public void handleResponse(Serializable response, int id) {
        System.out.println("Response from " + id + ": " + response);
        _total += ((Double)response).doubleValue();
    }
    public double getTotal() {
        return _total;
    }
    private double _total = 0;
}
```

Now we can invoke the `submit` method:

```
// Perform Asynchronous adds
AdderHandler handler = new AdderHandler();
for (int i=0; i < 10; i++) {
    s.submit("add", new Object[] { new Double(i), new Double(i) },
handler);
}
// Wait until all the invocations have returned
s.waitUntilInactive(0);
System.out.println("Total: " + handler.getTotal());
```

This code first creates a `ServiceInvocationHandler`, and then it calls `submit` several times. In order to wait for all the responses to arrive, the call to `waitUntilInactive` causes the current thread to wait until all outstanding requests have finished. The argument is a timeout value in milliseconds; an argument of zero means wait indefinitely.

# Using a GridServer-Generated Proxy

Using the Service API is easy and flexible, but there is an easier way to access Service implementations you write and deploy on GridServer. After registering the Service Type using the Administration Tool, a menu action becomes available on the **Services > Services > Service Types** page for creating client-side access proxies in either Java or C#.

There are several advantages to using proxies:

- **No manual coding required** — the proxy code itself does the argument passing and return value casting. It's a lot easier to use a class that looks like your Service!
- **Type-safe** — Argument and return types are checked at compile-time, instead of on the Engine at runtime.
- **Vendor Neutral** — Application code does not need to have compile-time dependency with any DataSynapse-specific libraries if the proxies are used. The proxies themselves make use of DataSynapse libraries, but do not expose this dependency to your application.

The following code demonstrates calling the `add` method on the proxy object:

```
// Create an instance of the Service Proxy
JavaAdderProxy adder = new JavaAdderProxy();
// Perform Synchronous add
double sum = adder.add(8.0, 10.0);
System.out.println("Result of Add: " + sum);
```

So what's going on behind the scenes? Here is a look at the `JavaAdderProxy` code for the method `add`:

```
public double add(double in0, double in1) throws Exception {
    return ((java.lang.Double) execute("add", new Object[] {new
java.lang.Double(in0), new java.lang.Double(in1)})).doubleValue();
}
```

If you want to call this method asynchronously (possibly to have many requests done in parallel) you can create a callback class that implements an interface found in the proxy. The callback interface defined in the proxy is found below:

```
public interface Callback extends ServiceBindingStub.AsyncCallback {
    public void handleResponse(Object response, int id);
```

```
    public void handleError(Exception e, int id);
}
```

Use this class in a fashion similar to that of example 3.3. Instead of implementing the `ServiceInvocationHandler` class you must implement the `JavaAdderProxy.Callback` class. The benefit to this approach, however, is that the code you write for this callback does not have to import any DataSynapse classes. All you need to do is import your proxy and use it.

When using a GridServer-generated proxy, asynchronous calls are performed by invoking an overloaded method that has the same name and arguments as the synchronous version, but adds an additional argument for the callback. The proxy also has a `waitUntilInactive` method to pause the current thread until all outstanding asynchronous calls have completed. The following example illustrates this:

```
// Perform Asynchronous adds
AdderCallback callback = new AdderCallback();
for (int i=0; i < 10; i++) {
    adder.add(i, i, callback);
}
// Wait until all the invocations have returned
adder.waitUntilInactive();
System.out.println("Total: " + callback.getTotal());
```

# Container Bindings and Service State

In this section we discuss the relationship between the container (GridServer Engine), the Service implementation (class or library), and operations that are exposed by the Service.

## About State

The GridServer technology allows Services to have a state—per-client or global—that is preserved by the container and is consistent across all physical Service instances, that is, all Engine-side instantiations of a client-side Service instance. The latter concept is important to understand: since many GridServer Engines can perform Service requests in parallel, the Service deployer must inform GridServer which methods are responsible for state and which are not. This way, GridServer can ensure that the object's state is up-to-date prior to servicing the next request. In addition to the state, the container must know which methods to expose as Service operations and which methods to call for initialization and destruction of the Service implementations in memory.

## Container-managed Lifecycle

Services hosted in the GridServer environment are virtualized—provisioned dynamically on Engines at the behest of the GridServer Manager to satisfy demand. In this model, Service initialization and destruction happens automatically, but proper start-up and clean-up operations can be performed on the Service by registering the right methods to call on the Service to accomplish these tasks. The following simple class demonstrates a Service that connects to a hypothetical database:

```
public class DBCalculator {
    public void initDBConnection(Properties connProps, int someVal) {
        _connection = new Connection(Properties props);
    }
    public Example calculate(String arg1, …) { … }
    public void close() {
        if (_connection != null) _connection.close();
    }
```

```
    private Connection _connection;
}
```

When a class is registered as a Service Type, it is possible to specify in the **ContainerBinding** section of the registry page which methods are used for initialization and destruction of the Service. In this example, `initDBConnection` is specified as the “initMethod” field and `close` is specified in the “destroyMethod” field. Since `initDBConnection` takes two arguments, a Properties object and an `int`, the client-side proxy class has a constructor with the same argument list:

```
public class DBCalculatorProxy {
    public DBCalculatorProxy(Properties connProps, int someVal) { … }
    …
}
```

The “close” method is listed as the `destroyMethod`, and is exposed on the client-side proxy as a method named “destroy.”

# Cancel Method

If a Service is canceled by the user or a Service request (task) is canceled, the managing container receives a message and calls a specified operation on the Service automatically. For a given Service, a method can be assigned to be called by the container under this cancellation event. Here’s a simple example:

```
public class DBCalculator {
    …
    public void stop() {
        _isStopped = true;
    }
    public Bar calculate(String arg1, …) {
        while (_isStopped != null) {
           // iterate over looping variable
        }
    }
    private volatile boolean _isStopped = false;
}
```

In this example, the `stop` method must be registered as the “cancelMethod”. If it is not possible to stop the operation in this way, GridServer allows cancelled Service operations to cause the entire Engine process to be killed and restarted automatically. The Service option is called `KILL_ENGINE_ON_CANCEL` and takes the value true or false.

# Stateful Service Operations

Now let us demonstrate how to manage state in the Service instances and how to instruct GridServer to behave properly under these operating conditions. First, let’s consider the following bond calculator class that holds state:

```
public class BondValuationService {
    public void addBonds(String[] bondIds) {…}
    public void setBonds (String[] bondIds) {…}
    public ValuationResults valueAllBonds(Scenario s) {…}
}
```

Both the `addBonds` and `setBonds` methods take a list of bond IDs. We assume that in these methods the actual object representations are constructed and the bond data is fetched from an appropriate source and loaded into the object representations. We further assume that there is a fair amount of latency and computation involved in these object constructions—this is why the writer of this Service is interested in keeping this stateful data present in the Service. There is also a stateless computing method that computes the valuations for these financial instruments under different market conditions or scenarios.

In this example, we see that there are two stateful methods—`addBonds` and `setBonds`. But their behavior is different: if `addBonds` is called, the new bonds are added to the existing collection, while the `setBonds` method replaces the existing collection. One can capture this distinction and ensure proper behavior of the GridServer deployment by setting the appropriate fields in the **ContainerBinding** section of the **Service Types** page. There is a field called `appendStateMethods` that can be set to the value `addBonds` and a field called `setStateMethods` that can be set to the value `setBonds`. More than one method can be listed, separating them by commas.

The client-side proxy class generated by GridServer looks quite similar to the actual Service:

```
public class BondValuationServiceProxy {
    public void addBonds(String[] bondIds) {…}
    public void setBonds (String[] bondIds) {…}
    public ValuationResults valueAllBonds(Scenario s) {…}
}
```

However, if you are using the Service API, you have to make the distinction in the API calls themselves. Notice the following use of the method `updateState`:

```
Service service = ServiceFactory.getInstance().createService
("bondCalculator");
service.updateState("addBonds", new Object[] { bondIds }, false);
// or
service.updateState("setBonds", new Object[] { bondIds }, true);
```

The last argument to the `updateState` method is a Boolean that indicates whether this state must replace the current state (true) or append the current state (false).

# Summary

- You can initialize and destroy Service state by writing methods in your Service that perform the required actions, and setting the `initMethod` and `destroyMethod` options to the names of the methods.
- The client-side proxy creates a constructor with the same method argument types as the `initMethod` that was specified in the Service Type Registry.
- The client-side proxy contains a destroy method that calls the method specified as the `destroyMethod` in the Service Type.
- The Service deployer must also specify which Service methods change state, and whether they append or set the state. These methods are called normally from a proxy, and via `updateState` using the Service API.

# TIBCO Documentation and Support Services

For information about this product, you can read the documentation, contact TIBCO Support, and join TIBCO Community.

## How to Access TIBCO Documentation

Documentation for TIBCO products is available on the TIBCO Product Documentation website, mainly in HTML and PDF formats.

The TIBCO Product Documentation website is updated frequently and is more current than any other documentation included with the product.

## Product-Specific Documentation

Documentation for TIBCO GridServer® is available on the TIBCO GridServer® Product Documentation page.

The following documents for this product can be found in the TIBCO Documentation site:

- TIBCO GridServer® Release Notes
- TIBCO GridServer® Installation
- TIBCO GridServer® Introducing TIBCO GridServer®
- TIBCO GridServer® Administration
- TIBCO GridServer® Developer's Guide
- TIBCO GridServer® Upgrade
- TIBCO GridServer® Security
- TIBCO GridServer® COM Integration Tutorial
- TIBCO GridServer® PDriver Tutorial
- TIBCO GridServer® Speedlink
- TIBCO GridServer® Service-Oriented Integration Tutorial

## How to Contact TIBCO Support

Get an overview of TIBCO Support. You can contact TIBCO Support in the following ways:

- For accessing the Support Knowledge Base and getting personalized content about products you are interested in, visit the TIBCO Support website.
- For creating a Support case, you must have a valid maintenance or support contract with TIBCO. You also need a user name and password to log in to TIBCO Support website. If you do not have a user name, you can request one by clicking **Register** on the website.

## How to Join TIBCO Community

TIBCO Community is the official channel for TIBCO customers, partners, and employee subject matter experts to share and access their collective experience. TIBCO Community offers access to Q&A forums, product wikis, and best practices. It also offers access to extensions, adapters, solution accelerators, and tools that extend and enable customers to gain full value from TIBCO products. In addition, users can submit and vote on feature requests from within the TIBCO Ideas Portal. For a free registration, go to TIBCO Community.

# Legal and Third-Party Notices



This software may be available on multiple operating systems. However, not all operating system platforms for a specific software version are released at the same time. See the readme file for the availability of this software version on a specific operating system platform.

THIS DOCUMENT IS PROVIDED “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS DOCUMENT COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THIS DOCUMENT. TIBCO SOFTWARE INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS DOCUMENT AT ANY TIME.

THE CONTENTS OF THIS DOCUMENT MAY BE MODIFIED AND/OR QUALIFIED, DIRECTLY OR INDIRECTLY, BY OTHER DOCUMENTATION WHICH ACCOMPANIES THIS SOFTWARE, INCLUDING BUT NOT LIMITED TO ANY RELEASE NOTES AND "READ ME" FILES.

This and other products of TIBCO Software Inc. may be covered by registered patents. Please refer to TIBCO's Virtual Patent Marking document (https://www.tibco.com/patents) for details.