Symphony Developer Tutorials

Platform Symphony
Version 5.1
April 2011

Platform Computing

Copyright

© 1994-2011 Platform Computing Corporation

All rights reserved.

Although the information in this document has been carefully reviewed, Platform Computing Corporation ("Platform") does not warrant it to be free of errors or omissions. Platform reserves the right to make corrections, updates, revisions or changes to the information in this document.

UNLESS OTHERWISE EXPRESSLY STATED BY PLATFORM, THE PROGRAM DESCRIBED IN THIS DOCUMENT IS PROVIDED "AS IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. IN NO EVENT WILL PLATFORM COMPUTING BE LIABLE TO ANYONE FOR SPECIAL, COLLATERAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, INCLUDING WITHOUT LIMITATION ANY LOST PROFITS, DATA, OR SAVINGS, ARISING OUT OF THE USE OF OR INABILITY TO USE THIS PROGRAM.

We'd like to hear from you

You can help us make this document better by telling us what you think of the content, organization, and usefulness of the information. If you find an error, or just want to make a suggestion for improving this document, please address your comments to doc@platform.com.

Your comments should pertain only to Platform documentation. For product support, contact support@platform.com.

Document redistribution and translation

This document is protected by copyright and you may not redistribute or translate it into another language, in part or in whole.

Internal redistribution

You may only redistribute this document internally within your organization (for example, on an intranet) provided that you continue to check the Platform Web site for updates and update your version of the documentation. You may not make it available to your organization over the Internet.

Trademarks

® LSF is a registered trademark of Platform Computing Corporation in the United States and in other jurisdictions.

™ ACCELERATING INTELLIGENCE, PLATFORM COMPUTING, PLATFORM SYMPHONY, PLATFORM JOB SCHEDULER, PLATFORM ISF, PLATFORM ENTERPRISE GRID ORCHESTRATOR, PLATFORM EGO, and the PLATFORM and PLATFORM LSF logos are trademarks of Platform Computing Corporation in the United States and in other jurisdictions.

® UNIX is a registered trademark of The Open Group in the United States and in other jurisdictions.

Linux® is the registered trademark of Linus Torvalds in the U.S. and other countries.

Microsoft is either a registered trademark or a trademark of Microsoft Corporation in the United States and/or other countries.

® Windows is a registered trademark of Microsoft Corporation in the United States and other countries.

Intel®, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Other products or services mentioned in this document are identified by the trademarks or service marks of their respective owners.

Third-party license agreements

http://www.platform.com/Company/third.part.license.htm

Third-party copyright notices

http://www.platform.com/Company/Third.Party.Copyright.htm

Contents

C++ Tutorials 5
Tutorial: Synchronous Symphony C++ client tutorial 5
Tutorial: SampleApp: Developing an asynchronous Symphony C++ client 13
Tutorial: SampleApp: Your first Symphony C++ service 22
Tutorial: SharingData: Developing a C++ client and service to share data among tasks 28
Java Tutorials 34
Tutorial: SampleApp: Your first synchronous Symphony Java client 34
Tutorial: SampleApp: Developing an asynchronous Symphony Java client 44
Tutorial: SampleApp: Your first Symphony Java service 53
Tutorial: SharingData: Developing a Java client and service to share data among tasks 58
.NET Tutorials 67
Tutorial: SampleApp: Your first synchronous Symphony C# client and service 67
Tutorial: SampleApp: Developing an asynchronous Symphony C# client 77
Tutorial: SharingData: Developing a C# client and service to share data among tasks 83
Cross-language Tutorials 91
Tutorial: CrossLanguage: Developing cross-language clients and services 91
COM Tutorial 105
Tutorial: Developing a COM API client 105
Eclipse Tutorial 117
Tutorial: Developing a Symphony application with Eclipse 117
Visual Studio Tutorial 135
Tutorial: On-boarding a Symphony application with Visual Studio 135

C++ Tutorials

Tutorial: Synchronous Symphony C++ client tutorial

Goal

This tutorial guides you through the process of building, packaging, deploying, and running the hello grid sample client and service. It also walks you through the sample client application code.

You learn the minimum amount of code that you need to create a client.

At a glance

Before you begin, ensure you have installed and started Platform Symphony DE. You complete the following tasks:

1. Build the sample client and service
2. Package the sample service
3. Add the application
4. Run the sample client and service
5. Walk through the code

Build the sample client and service

On Windows

You can build client application and service samples at the same time.

1. In `%SOAM_HOME%\5.1\samples\CPP\SampleApp`, locate workspace file `sampleCPP_vc6.dsw`, or one of the Visual Studio solution files.
2. Load the file into Visual Studio and build it.

On Linux

You can build client application and service samples at the same time.

1. Change to the `conf` directory under the directory in which you installed Symphony DE.

   For example, if you installed Symphony DE in `/opt/symphonyDE/DE51`, go to `/opt/symphonyDE/DE51/conf`.
2. Source the environment:

   - For `csh`, enter

     ```
     source cshrc.soam
     ```
   - For `bash`, enter

     ```
     . profile.soam
     ```
3. Compile using the Makefile located in `$SOAM_HOME/5.1/samples/CPP/SampleApp`:

make

Package the sample service

On Windows

To deploy the service, you first need to package it.

1. Go to the directory in which the compiled samples are located.

   cd %SOAM_HOME%\5.1\samples\CPP\SampleApp\Output

2. Create the service package by compressing the service executable into a zip file.

   gzip SampleServiceCPP.exe

   You have now created your service package `SampleServiceCPP.exe.gz`.

On Linux

To deploy the service, you first need to package it.

1. Change to the directory in which the compiled samples are located:

   cd $SOAM_HOME/5.1/samples/CPP/SampleApp/Output/

2. Create the service package by archiving and compressing the service executable file:

   tar -cvf SampleServiceCPP.tar SampleServiceCPP

   gzip SampleServiceCPP.tar

   You have now created your service package `SampleServiceCPP.tar.gz`.

Add the application

When you add an application through the DE PMC, you must use the Add Application wizard. This wizard defines a consumer location to associate with your application, deploys your service package, and registers your application. After completing the steps with the wizard, your application should be ready to use.

1. In the DE PMC, click Symphony Workload > Configure Applications.

   The Applications page displays.

2. Select Global Actions > Add/Remove Applications.

   The Add/Remove Application page displays.

3. Select Add an application, then click Continue.

   The Adding an Application page displays.

4. Select Use existing profile and add application wizard. Click Browse and navigate to your application profile.

5. Select your application profile xml file, then click Continue.

   For SampleApp, you can find your profile in the following location:

   - Windows—`%SOAM_HOME%\5.1\samples\CPP\SampleApp\SampleApp.xml`
   - Linux—`$SOAM_HOME/5.1/samples/CPP/SampleApp/SampleApp.xml`

The Service Package location window displays.

6. Browse to the service package you created in .gz or tar.gz format and select it. Click Continue.

   The Confirmation window displays.

7. Review your selections, then click Confirm.

   The window displays indicating progress. Your application is ready to use.

8. Click Close.

   The window closes and you are now back in the Platform Management Console. Your new application is displayed as enabled.

Run the sample client and service

On Windows

To run the service, you run the client application. The service a client application uses is specified in the application profile.

1. Run the client application:

   %SOAM_HOME%\5.1\samples\CPP\SampleApp\Output\SyncClient.exe

   You should see output on the command line as work is submitted to the system.

   The client starts and the system starts the corresponding service. The client displays messages indicating that it is running.

On Linux

1. Run the client application:

   $SOAM_HOME/5.1/samples/CPP/SampleApp/Output/SyncClient

   You should see output on the command line as work is submitted to the system.

   The client starts and the system starts the corresponding service. The client displays messages indicating that it is running.

Review and understand the samples

You review the sample client application code to learn how you can create a synchronous client application.

Locate the code samples

Operating System	Files	Location of Code Sample
Windows	Client	`%SOAM_HOME%\5.1\samples\CPP\SampleApp\SyncClient`
	Message object	`%SOAM_HOME%\5.1\samples\CPP\SampleApp\Common`
	Service code	`%SOAM_HOME%\5.1\samples\CPP\SampleApp\Service`
	Application profile	The service required to compute the input data along with additional application parameters are defined in the application profile: `%SOAM_HOME%\5.1\samples\CPP\SampleApp\SampleApp.xml`
	Output directory	`%SOAM_HOME%\5.1\samples\CPP\SampleApp\Output\`
Linux	Client	`$SOAM_HOME/5.1/samples/CPP/SampleApp/SyncClient`
	Message object	`$SOAM_HOME/5.1/samples/CPP/SampleApp/Common`
	Service code	`$SOAM_HOME/5.1/samples/CPP/SampleApp/Service`
	Application profile	The service required to compute the input data along with additional application parameters are defined in the application profile: `$SOAM_HOME/5.1/samples/CPP/SampleApp/SampleApp.xml`
	Output directory	`$SOAM_HOME/5.1/samples/CPP/SampleApp/Output/`

What the sample does

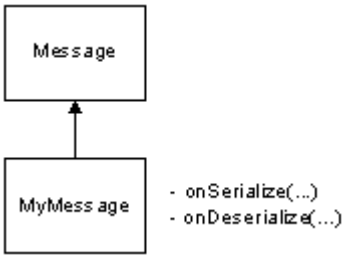
The client application sample opens a session and sends 10 input messages, and retrieves the results. The client application is a synchronous client that sends input and blocks the output until all the results are returned.

The service takes input data sent by client applications, returns the input data you have sent and replies "Hello Client!!"

Review the sample code

Input and output: declare the message object

Your client application needs to handle data that it sends as input, and output data that it receives from the service.



Tip:

Client applications and services share the same message class.

In `MyMessage.h`:

- We declare the `MyMessage` class
- We define serialization methods for input and output messages
- We implement methods to handle the data

Note:

For this example, we have defined the same class for input and output messages. However, you can define separate classes for input and output messages.

```
#pragma once
#include "soam.h"
class MyMessage :
    public soam::Message
{
public:
    MyMessage();
    MyMessage(int i, bool isSync, char* str);
    virtual ~MyMessage(void);
    void onSerialize(
        /*[in]*/ soam::OutputStreamPtr &stream) throw (soam::SoamException);
    void onDeserialize(
    /*[in]*/ soam::InputStreamPtr &stream) throw (soam::SoamException);
// accessors
public:
    int getInt() const{return m_int;}
    void setInt(int _int) {m_int = _int;}
    const char* getString() {return m_string;}
    void setString(const char* str) {freeString(m_string); m_string = copyString(str);}
    bool getIsSync() const{return (m_isSync != 0);}
    void setIsSync(bool isSync) {m_isSync = isSync;}
private:
    char* copyString(const char* strSource);
    void freeString(char* strToFree);
private:
    int m_int;
    bool m_isSync;
    char* m_string;
};
```

Implement the MyMessage object

Once your message object is declared, implement handlers for serialization and deserialization.

In `MyMessage.cpp`, we implement methods to handle the data. For data types that are supported by Symphony DE, see the appropriate API reference.

Note:

If you already have an application with a message object that is serialized, you can pass a binary blob through the DefaultBinaryMessage class.

```
#include "stdafx.h"
#include <string.h>
#include "MyMessage.h"
#include "soam.h"
using namespace soam;
MyMessage::MyMessage()
{
    m_int = 0;
    m_string = copyString("");
}
MyMessage::MyMessage(int i, bool isSync, char* str)
{
    m_int = i;
    m_isSync = isSync;
    m_string = copyString(str);
}
MyMessage::~MyMessage(void)
{
    freeString(m_string);
}void MyMessage::onSerialize(OutputStreamPtr &stream) throw (SoamException)
{
    stream->write(m_int);
    stream->write(m_isSync);
    stream->write(m_string);
}void MyMessage::onDeserialize(InputStreamPtr &stream) throw (SoamException)
{
    stream->read(m_int);
    stream->read(m_isSync);
    freeString(m_string);
    stream->read(m_string);
}char* MyMessage::copyString(const char* strSource)
{
    SOAM_ASSERT(0 != strSource);
    size_t len = strlen(strSource);
    char* newString = new char[len+1];
    SOAM_ASSERT(0 != newString);
    strcpy(newString, strSource);
    return newString;
}
void MyMessage::freeString(char* strToFree)
{
    if (0 != strToFree)
    {
        delete []strToFree;
    }
}
```

Initialize the client

In `SyncClient.cpp`, when you initialize, you initialize the Symphony client infrastructure. You initialize once per client.

Important:

Initialization is required. Otherwise, API calls fail.

```
...
    try
    {
        // Initialize the API
        SoamFactory::initialize();
...
```

Connect to an application

To send data to be calculated in the form of input messages, you connect to an application.

You specify an application name, a user name, and password. The application name must match that defined in the application profile.

For Symphony DE, there is no security checking and login credentials are ignored—you can specify any user name and password. Security checking is done however, when your client application submits workload to the actual grid.

The default security callback encapsulates the callback for the user name and password.

Tip:

When you connect, a connection object is returned.

```
...
  // Set up application specific information to be supplied to Symphony
     char appName[]="SampleAppCPP";
  // Set up application authentication information using the default security provider
     DefaultSecurityCallback securityCB("Guest", "Guest");
  // Connect to the specified application
     ConnectionPtr conPtr = SoamFactory::connect(appName, &securityCB);
  // Retrieve and print our connection ID
     cout << "connection ID=" << conPtr->getId() << endl;
...
```

Create a session to group tasks

A session is a way of logically grouping tasks that are sent to a service for execution. The tasks are sent and received synchronously.

When creating a session, you need to specify the session attributes by using the SessionCreationAttributes object. In this sample, we create a SessionCreationAttributes object called attributes and set three parameters in the object.

The first parameter is the session name. This is optional. The session name can be any descriptive name you want to assign to your session. It is for information purposes, such as in the command-line interface.

The second parameter is the session type. The session type is optional. If you leave this parameter blank " " or do not set a session type, system default values are used for session attributes. If you specify a session type in the client application, you must also configure the session type in the application profile—the session type name in your application profile and session type you specify in the client must match. If you use an incorrect session type in the client and the specified session type cannot be found in the applicatin profile, an exception is thrown to the client.

The third parameter is the session flag, which we specify as ReceiveSync. You must specify it as shown. This indicates to Symphony that this is a synchronous session.

We pass the attributes object to the createSession() method, which returns a pointer to the session.

```
       // Set up session creation attributes
       SessionCreationAttributes attributes;
       attributes.setSessionName("mySession");
       attributes.setSessionType("ShortRunningTasks");
       attributes.setSessionFlags(Session::ReceiveSync);
       // Create a synchronous session
       SessionPtr sesPtr = conPtr->createSession(attributes);
```

Send input data to be processed

In this step, we create 10 input messages to be processed by the service. When a message is sent, a task input handle is returned. This task input handle contains the ID for the task that was created for this input message.

```
int tasksToSend = 10;
       for (int taskCount = 0; taskCount < tasksToSend; taskCount++)
       {
           // Create a message
           char hello[]="Hello Grid !!";
           MyMessage inMsg(taskCount, true, hello);
           // Create task attributes
           TaskSubmissionAttributes attrTask;
           attrTask.setTaskInput(&inMsg);
                     // send it
           TaskInputHandlePtr input = sesPtr->sendTaskInput(attrTask);
           // Retrieve and print task ID
           cout << "task submitted with ID : " << input->getId() << endl;
       }
...
```

Retrieve output

Pass the number of tasks to the fetchTaskOutput() method to retrieve the output messages that were produced by the service. This method blocks until the output for all tasks is retrieved. The return value is an enumeration that contains the completed task results. Iterate through the task results and extract the messages using the populateTaskOutput() method. Display the task ID and the results from the output message.

```
       // Now get our results - will block here until all tasks retrieved
       EnumItemsPtr enumOutput = sesPtr->fetchTaskOutput(tasksToSend);
       // Inspect results
       TaskOutputHandlePtr output;
       while(enumOutput->getNext(output))
       {
           // Check for success of task
           if (true == output->isSuccessful())
           {
               // Get the message returned from the service
               MyMessage outMsg;
               output->populateTaskOutput(&outMsg);
               // Display content of reply
               cout << "Task Succeeded [" <<  output->getId() << "]" << endl;
               cout << outMsg.getResult() << endl;
           }
           else
           {
               // Get the exception associated with this task
               SoamExceptionPtr ex = output->getException();
               cout << "Task Failed : " << ex->what() << endl;
           }
       }
```

Catch exceptions

Any exceptions thrown take the form of SoamException. Catch all Symphony exceptions to know about exceptions that occurred in the client application, service, and middleware.

The sample code above catches exceptions of type SoamException.

```
catch(SoamException& exp)
{
// Report exception
cout << "exception caught ... " << exp.what() << endl;
}
```

Uninitialize

Always uninitialize the client API at the end of all API calls. If you do not call uninitialize, the client API is in an undefined state, resources used by the client are held indefinitely, and there is no guarantee your client will be stable.

Important:

Once you uninitialize, all objects become invalid. For example, you can no longer create a session or send an input message.

```
    // uninitialize the API
    // This is the only means to ensure proper shutdown
    // of the interaction between the client and the system.
    SoamFactory::uninitialize();
...
```

Tutorial: SampleApp: Developing an asynchronous Symphony C++ client

Goal

The purpose of an asynchronous client is to get the output as soon as it is available. The client thread does not need to be blocked once the input data is sent and can perform other actions.

In this tutorial, you learn how to convert a synchronous client into asynchronous.

At a glance

Before you begin, ensure you have installed and started Platform Symphony DE. You complete the following tasks:

1. Build the sample client and service
2. Package the sample service
3. Add the application
4. Run the sample client and service
5. Walk through the code

Build the sample client and service

On Windows

You can build client application and service samples at the same time.

1. In `%SOAM_HOME%\5.1\samples\CPP\SampleApp`, locate workspace file `sampleCPP_vc6.dsw`, or one of the Visual Studio solution files.

2. Load the file into Visual Studio and build it.

On Linux

You can build client application and service samples at the same time.

1. Change to the `conf` directory under the directory in which you installed Symphony DE.

   For example, if you installed Symphony DE in `/opt/symphonyDE/DE51`, go to `/opt/symphonyDE/DE51/conf`.
2. Source the environment:
   - For `csh`, enter

     ```
     source cshrc.soam
     ```
   - For `bash`, enter

     ```
     . profile.soam
     ```
3. Compile using the Makefile located in `$SOAM_HOME/5.1/samples/CPP/SampleApp`:

   ```
   make
   ```

Package the sample service

On Windows

To deploy the service, you first need to package it.

1. Go to the directory in which the compiled samples are located.

   ```
   cd %SOAM_HOME%\5.1\samples\CPP\SampleApp\Output\
   ```
2. Create the service package by compressing the service executable into a zip file.

   ```
   gzip SampleServiceCPP.exe
   ```

   You have now created your service package `SampleServiceCPP.exe.gz`.

On Linux

To deploy the service, you first need to package it.

1. Change to the directory in which the compiled samples are located:

   ```
   cd $SOAM_HOME/5.1/samples/CPP/SampleApp/Output/
   ```
2. Create the service package by compressing the service executable into a tar file:

   ```
   tar -cvf SampleServiceCPP.tar SampleServiceCPP
   ```

   ```
   gzip SampleServiceCPP.tar
   ```

   You have now created your service package `SampleServiceCPP.tar.gz`.

Add the application

When you add an application through the DE PMC, you must use the Add Application wizard. This wizard defines a consumer location to associate with your application, deploys your service package, and registers your application. After completing the steps with the wizard, your application should be ready to use.

1. In the DE PMC, click Symphony Workload > Configure Applications.

   The Applications page displays.
2. Select Global Actions > Add/Remove Applications.

   The Add/Remove Application page displays.
3. Select Add an application, then click Continue.

   The Adding an Application page displays.
4. Select Use existing profile and add application wizard. Click Browse and navigate to your application profile.
5. Select your application profile xml file, then click Continue.

   For SampleApp, you can find your profile in the following location:

   - Windows—`%SOAM_HOME%\5.1\samples\CPP\SampleApp\SampleApp.xml`
   - Linux—`$SOAM_HOME/5.1/samples/CPP/SampleApp/SampleApp.xml`

   The Service Package location window displays.
6. Browse to the service package you created in .gz or tar.gz format and select it, then, click Continue.

   The Confirmation window displays.
7. Review your selections, then click Confirm.

   The window displays indicating progress. Your application is ready to use.
8. Click Close.

   The window closes and you are now back in the Platform Management Console. Your new application is displayed as enabled.

Run the sample client and service

On Windows

To run the service, you run the client application. The service a client application uses is specified in the application profile.

1. Run the client application:

   %SOAM_HOME%\5.1\samples\CPP\SampleApp\Output\AsyncClient.exe

   You should see output on the command line as work is submitted to the system.

   The client starts and the system starts the corresponding service. The client displays messages indicating that it is running.

On Linux

1. Run the client application:

   $SOAM_HOME/5.1/samples/CPP/SampleApp/Output/AsyncClient

   You should see output on the command line as work is submitted to the system.

   The client starts and the system starts the corresponding service. The client displays messages indicating that it is running.

Walk through the code

You review the sample client application code to learn how you can understand the differences between a synchronous client and an asynchronous client.

Locate the code samples

Operating System	Files	Location of Code Sample
Windows	Client	`%SOAM_HOME%\5.1\samples\CPP\SampleApp\AsyncClient`
	Message object	`%SOAM_HOME%\5.1\samples\CPP\SampleApp\Common`
	Service code	`%SOAM_HOME%\5.1\samples\CPP\SampleApp\Service`
	Application profile	The service required to compute the input data along with additional application parameters are defined in the application profile: `%SOAM_HOME%\5.1\samples\CPP\SampleApp\SampleApp.xml`
	Output directory	`%SOAM_HOME%\5.1\samples\CPP\SampleApp\Output\`
Linux	Client	`$SOAM_HOME/5.1/samples/CPP/SampleApp/AsyncClient`
	Message object	`$SOAM_HOME/5.1/samples/CPP/SampleApp/Common`
	Service code	`$SOAM_HOME/5.1/samples/CPP/SampleApp/Service`
	Application profile	The service required to compute the input data along with additional application parameters are defined in the application profile: `$SOAM_HOME/5.1/samples/CPP/SampleApp/SampleApp.xml`
	Output directory	`$SOAM_HOME/5.1/samples/CPP/SampleApp/Output/`

What the sample does

The client application sample sends 10 input messages with the data Hello Grid!! and retrieves the results.

Results are returned asynchronously with a callback interface provided by the client to the API. Methods on this interface are called from threads within the API when certain events occur. In the sample, the events are:

- When there is an error at the session level
- When results return from Symphony

Considerations for asynchronous clients

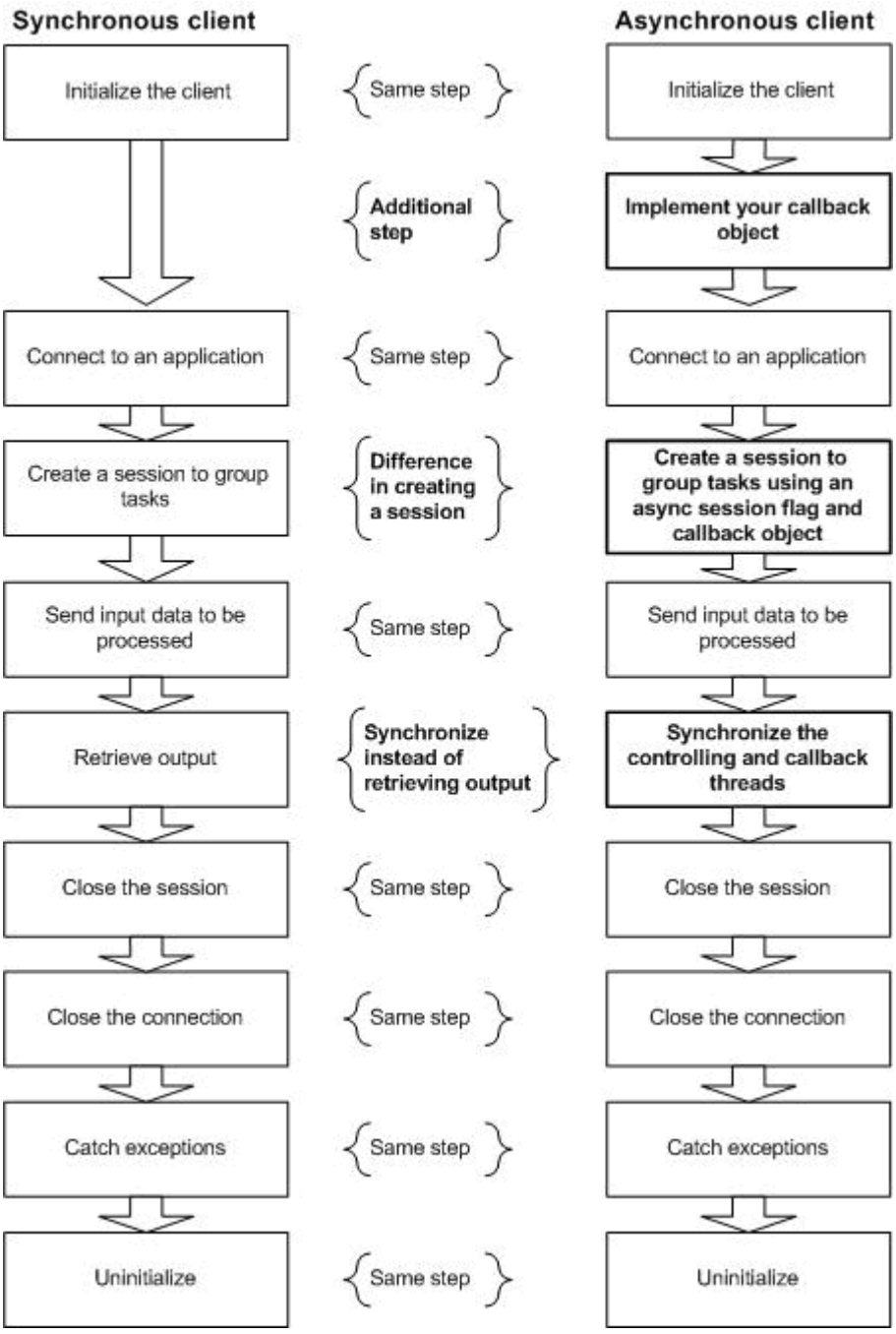
Synchronization	Because results can come back at any time, it is probable that your callback code needs synchronization between the callback thread and the controlling thread. The controlling thread needs to know when work is complete.
Order of results	Results are not sent back in order. If order of results is important, the client application must sort the results.

Code differences between synchronous and asynchronous clients

An asynchronous client is very similar to a synchronous client. The only differences are:

- You need to specify a callback when creating a session
- You specify a different flag to indicate asynchronous when you create a session
- Retrieval of replies

Let us look at the steps to create synchronous and asynchronous clients and highlight the differences. Steps in bold indicate differences. Everything else is the same as the synchronous client.



Declare the message object and implement

As in the synchronous client tutorial, declare the message object and implement your own message object.

If you have not done so already, take a look at the synchronous client application tutorial Your First Synchronous Symphony C++ Client for details on the Message object, specifically:

- Input and output: declare the message object
- Implement the MyMessage object

Declare and implement your callback object

Perform this step after declaring the Message object and implementing the MyMessage object.

In `MyCallback.h`, we create our own callback class from the SessionCallback class, and we implemented onResponse() to retrieve the output for each input message that we send.

Note:

- onResponse() is called every time a task completes and output is returned to the client. The task output handle allows the client code to manipulate the output.
- isSuccessful() checks whether there is output to retrieve.
- If there is output to retrieve, populateTaskOutput() gets the output. Once results return, we print them to standard output and return.

```
#include "soam.h"
using namespace soam;
using namespace std;
#ifndef WIN32
#include <pthread.h>
#else
#include <windows.h>
#endif
class MySessionCallback :
    public SessionCallback
{
    public:
        MySessionCallback()
            :m_tasksReceived(0), m_exception(false)
        {
#ifndef WIN32
            pthread_mutexattr_t attr;
            pthread_mutexattr_init( &attr );
            pthread_mutexattr_settype( &attr, PTHREAD_MUTEX_RECURSIVE );
            pthread_mutex_init( &m_mutex, &attr );
            pthread_mutexattr_destroy( &attr );
#else
            InitializeCriticalSection(&m_criticalSection);
#endif
            cout << "Callback created ... " << endl;
        }
virtual ~MySessionCallback()
        {
#ifndef WIN32
            pthread_mutex_destroy( &m_mutex );
#else
            DeleteCriticalSection(&m_criticalSection);
#endif
        }
```

```
//////////////////////////////////////////////////////
// This handler is called once any exception occurs
// within the scope of the session.
// ==================================================
virtual void onException(SoamException &exception) throw()
{
cout << "An exception occured on the callback.\nDetails : " <<  exception.what() << endl;
#ifndef WIN32
                pthread_mutex_lock( &m_mutex);
#else
                EnterCriticalSection(&m_criticalSection);
#endif
                m_exception = true;
#ifndef WIN32
                pthread_mutex_unlock( &m_mutex);
#else
                LeaveCriticalSection(&m_criticalSection);
#endif
}
//////////////////////////////////////////////////////
// This handler is called once a message is returned
// from the system when a task completes.
// ==================================================
void onResponse(TaskOutputHandlePtr &output) throw()
        {
            try
            {
                // check for success of task
                if (true == output->isSuccessful())
                {                       // get the message returned from the service
                    MyMessage outMsg;
                    output->populateTaskOutput(&outMsg);
                    // display content of reply
                    cout << "Task Succeeded [" <<  output->getId() << "]" << endl;
                    cout << "Integer Value : " << outMsg.getInt() << endl;
                    cout << outMsg.getString() << endl;
                }
                else
                {
                    // get the exception associated with this task
                    SoamExceptionPtr ex = output->getException();
                    cout << "Task Failed : " << ex->what() << endl;
                }
            }
            catch(SoamException &exception)
            {
                cout << "Exception occured in OnResponse() : " << exception.what() << endl;
            }
```

```
// Update counter used to synchronize the controlling thread
// with this callback object
#ifndef WIN32
                pthread_mutex_lock( &m_mutex);
#else
                EnterCriticalSection(&m_criticalSection);
#endif
                ++m_tasksReceived;
#ifndef WIN32
                pthread_mutex_unlock( &m_mutex);
#else
                LeaveCriticalSection(&m_criticalSection);
#endif
        }

        inline long getReceived()
        {
            return m_tasksReceived;
        }
        inline bool getDone()
        {
            return m_exception;
        }

private:
#ifndef WIN32
        pthread_mutex_t m_mutex;
#else
        CRITICAL_SECTION m_criticalSection;
#endif
        long m_tasksReceived;
        bool m_exception;
};
```

Create a session to group tasks

In `AsyncClient.cpp`, perform this step after you have connected to the application.

When creating an asynchronous session, you need to specify the session attributes by using the SessionCreationAttributes object. In this sample, we create a SessionCreationAttributes object called attributes and set four parameters in the object.

The first parameter is the session name. This is optional. The session name can be any descriptive name you want to assign to your session. It is for information purposes, such as in the command line interface.

The second parameter is the session type. The session type is optional. You can leave this parameter blank or not make the API call at all. When you do this, system default values are used for your session.

The third parameter is the session flag, which we specify as ReceiveAsync. You must specify it as shown. This indicates to Symphony that this is an asynchronous session.

The fourth parameter is the callback object.

We pass the attributes object to the createSession() method, which returns a pointer to the session.

```
...
        // Create session callback
        MySessionCallback  myCallback;
        // Set up session creation attributes
        SessionCreationAttributes attributes;
        attributes.setSessionName("mySession");
        attributes.setSessionType("ShortRunningTasks");
        attributes.setSessionFlags(Session::ReceiveAsync);
        attributes.setSessionCallback(&myCallback);
        // Create an asynchronous session
        SessionPtr sesPtr = conPtr->createSession(attributes);
...
```

Synchronize the controlling and callback threads

Perform this step after sending the input data to be processed.

Since our client is asynchronous, we need to synchronize the controlling thread and the callback thread. In this example, the controlling thread blocks until all replies have come back.

```
...
// Now wait until all replies have been received asynchronously
// by our callback ... for illustrative purposes we will poll
// until all replies are in.
while ((myCallback.getReceived() < tasksToSend) && !myCallback.getDone())
 {
   ourSleep(2);
 }
...
```

Tutorial: SampleApp: Your first Symphony C++ service

Goal

This tutorial guides you through the process of building and running a service, then walks you through the sample service code.

You learn the minimum amount of code that you need to create a service.

At a glance

Before you begin, ensure you have installed and started Platform Symphony DE.

1. Build the sample client and service
2. Package the sample service
3. Add the application
4. Run the sample client and service
5. Walk through the code

Build the sample client and service

On Windows

You can build client application and service samples at the same time.

1. In %SOAM_HOME%\5.1\samples\CPP\SampleApp, locate workspace file `sampleCPP_vc6.dsw`, or one of the Visual Studio solution files.
2. Load the file into Visual Studio and build it.

On Linux

You can build client application and service samples at the same time.

1. Change to the `conf` directory under the directory in which you installed Symphony DE.

For example, if you installed Symphony DE in `/opt/symphonyDE/DE51`, go to `/opt/symphonyDE/DE51/conf`.

2. Source the environment:
   - For `csh`, enter

     source cshrc.soam
   - For `bash`, enter

     . profile.soam
3. Compile using the Makefile located in `$SOAM_HOME/5.1/samples/CPP/SampleApp`:

   make

Package the sample service

On Windows

To deploy the service, you first need to package it.

1. Go to the directory in which the compiled samples are located.

   cd %SOAM_HOME%\5.1\samples\CPP\SampleApp\Output
2. Create the service package by compressing the service executable into a zip file.

   gzip SampleServiceCPP.exe

   You have now created your service package `SampleServiceCPP.exe.gz`.

On Linux

To deploy the service, you first need to package it.

1. Change to the directory in which the compiled samples are located:

   cd $SOAM_HOME/5.1/samples/CPP/SampleApp/Output/
2. Create the service package by compressing the service executable into a tar file:

   tar -cvf SampleServiceCPP.tar SampleServiceCPP

   gzip SampleServiceCPP.tar

   You have now created your service package `SampleServiceCPP.tar.gz`.

Add the application

When you add an application through the DE PMC, you must use the Add Application wizard. This wizard defines a consumer location to associate with your application, deploys your service package, and registers your application. After completing the steps with the wizard, your application should be ready to use.

1. In the DE PMC, click Symphony Workload > Configure Applications.

   The Applications page displays.
2. Select Global Actions > Add/Remove Applications.

   The Add/Remove Application page displays.
3. Select Add an application, then click Continue.

The Adding an Application page displays.

4. Select Use existing profile and add application wizard. Click Browse and navigate to your application profile.
5. Select your application profile xml file, then click Continue.

   For SampleApp, you can find your profile in the following location:

   - Windows—`%SOAM_HOME%\5.1\samples\CPP\SampleApp\SampleApp.xml`
   - Linux—`$SOAM_HOME/5.1/samples/CPP/SampleApp/SampleApp.xml`

   The Service Package location window displays.
6. Browse to the service package you created in .gz or tar.gz format and select it, then, click Continue.

   The Confirmation window displays.
7. Review your selections, then click Confirm.

   The window displays indicating progress. Your application is ready to use.
8. Click Close.

   The window closes and you are now back in the Platform Management Console. Your new application is displayed as enabled.

Run the sample client and service

On Windows

To run the service, you run the client application. The service a client application uses is specified in the application profile.

1. Run the client application:

   %SOAM_HOME%\5.1\samples\CPP\SampleApp\Output\SyncClient.exe

   You should see output on the command line as work is submitted to the system.

   The client starts and the system starts the corresponding service. The client displays messages indicating that it is running.

On Linux

1. Run the client application:

   $SOAM_HOME/5.1/samples/CPP/SampleApp/Output/SyncClient

   You should see output on the command line as work is submitted to the system.

   The client starts and the system starts the corresponding service. The client displays messages indicating that it is running.

Walk through the code

You review the sample service code to learn how you can create a service.

Locate the code samples

Operating System	Files	Location of Code Sample
Windows	Client	`%SOAM_HOME%\5.1\samples\CPP\SampleApp\SyncClient`
	Message object	`%SOAM_HOME%\5.1\samples\CPP\SampleApp\Common`
	Service code	`%SOAM_HOME%\5.1\samples\CPP\SampleApp\Service`
	Application profile	The service required to compute the input data along with additional application parameters are defined in the application profile: `%SOAM_HOME%\5.1\samples\CPP\SampleApp\SampleApp.xml`
	Output directory	`%SOAM_HOME%\5.1\samples\CPP\SampleApp\Output`
Linux	Client	`$SOAM_HOME/5.1/samples/CPP/SampleApp/SyncClient`
	Message object	`$SOAM_HOME/5.1/samples/CPP/SampleApp/Common`
	Service code	`$SOAM_HOME/5.1/samples/CPP/SampleApp/Service`
	Application profile	The service required to compute the input data along with additional application parameters are defined in the application profile: `$SOAM_HOME/5.1/samples/CPP/SampleApp/SampleApp.xml`
	Output directory	`$SOAM_HOME/5.1/samples/CPP/SampleApp/Output/`

What the sample does

The service takes input data sent by client applications, returns the input data you have sent and replies "Hello Client !!"

Input and output: declare and implement the Message object:

Your service needs to handle data that it receives as input, and generate output data that can be sent back to the client application.

Note:

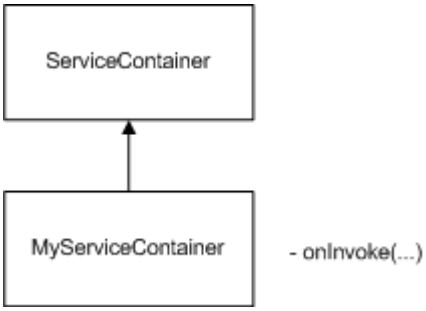
Client applications and services share the same message class. You do not need to create a different message class. In our example, we have created a common directory for code that is shared by client and service. Use the Message object declared and implemented by the client application.

If you have not done so already, take a look at the synchronous client application tutorial for details on the Message object.

- Input and output: declare the message object:
- Implement the MyMessage object:

Define a service container:

For a service to be managed by Symphony, it needs to be in a container object. This is the service container.



In `SampleService.cpp`, we inherited from the ServiceContainer class.

```
#include "stdafx.h"
#include <stdio.h>
#include "soam.h"
#include "MyMessage.h"
using namespace soam;
using namespace std;
class MyServiceContainer : public ServiceContainer
```

Process the input:

Symphony calls onInvoke() on the service container once per task. Once you inherit from the ServiceContainer class, implement handlers so that the service can function properly. This is where the calculation is performed.

To gain access to the data set for the client, you must present an instance of the message object to the populateTaskInput() method on the task context.

The task context contains all information and functionality that is available to the service during an onInvoke() call in relation to the task that is being processed.

Important:

Services are virtualized. As a result, a service should not read from stdin or write to stdout. Services can, however, read from and write to files that are accessible to all compute hosts.

You present to populateTaskInput() the message object that comes from the client application. During this call, the data sent from the client is used to populate the message object.

```
{
public:
    virtual void onInvoke (TaskContextPtr& taskContext)
    {
        // get the input that was sent from the client
        MyMessage inMsg;
        taskContext->populateTaskInput(inMsg);
        // We simply echo the data back to the client
        MyMessage outMsg;
        outMsg.setInt(inMsg.getInt());
        std::string str="you sent : ";
        str += inMsg.getString();
        str += "\nwe replied : Hello Client !!\n>>> ";
        if (inMsg.getIsSync())
        {
            str += "Synchronously.\n";
        }
        else
        {
            str += "Asynchronously.\n";
        }
        outMsg.setString(str.c_str());
        // set our output message
        taskContext->setTaskOutput(outMsg);
    }
};
```

Run the container:

The service is implemented within an executable. At a minimum, we need to create within our main function an instance of the service container and run it.

Note that your service is started by parameters.

```
int main(int argc, char* argv[])
{
    // return value of our service program
    int retVal = 0;
    try
    {
        // Create the container and run it
        MyServiceContainer myContainer;
        myContainer.run();
    }
```

Catch exceptions:

Catch exceptions in case the container fails to start running.

```
catch(SoamException& exp)
    {
        // report exception to stdout
        cout << "exception caught ... " << exp.what() << endl;
retVal = -1;
    }
    // NOTE: Although our service program will return an overall
    // failure or success code it will always be ignored in the
    // current revision of the middleware.
    // The value being returned here is for consistency.
    return retVal;
}
```

Tutorial: SharingData: Developing a C++ client and service to share data among tasks

Goal

This tutorial walks you through the sample common data object code, then describes how to use different data objects for input, output, and common data. How to develop a client application and service to share data among all tasks in a session. The data is shared by all invocations of tasks within the same session.

At a glance

Before you begin, ensure you have installed and started Platform Symphony Developer Edition. You should also have completed the tutorial Your First Synchronous Symphony C++ Client. When you are ready, do the following:

1. Build the sample client and service
2. Package the sample service
3. Add the application
4. Run the sample client and service
5. Walk through the code

When to use common data

Common data is data that can be made available to service instances for the duration of a session.

Use common data when you need to set up the initial state of a service, and you only want to do it once, not on every task. Common data is useful for passing data from a client to a service. The service loads the data when the session is created.

You can use common data, for example, to set the environment in the service that is common to all tasks in a session. This way you only need to set the environment once, when the session is created.

Symphony attempts to use the same service instance for all tasks in a session. A service instance is made available to other sessions only when session workload completes, a session is closed or aborted, or when another session of higher priority is assigned the service instance.

Build the sample client and service

On Windows

You can build client application and service samples at the same time.

1. Load the workspace file `sharing_data_vc6.dsw`, or one of the Visual Studio solution files into Visual Studio and build it.

On Linux

You can build client application and service samples at the same time.

1. Change to the `conf` directory under the directory in which you installed Symphony DE.

For example, if you installed Symphony Developer Edition in `/opt/symphonyDE/DE51`, go to `/opt/symphonyDE/DE51/conf`.

2. Source the environment:
   - For `csh`, enter

     ```
     source cshrc.soam
     ```
   - For `bash`, enter

     ```
     . profile.soam
     ```
3. Compile using the Makefile located in `$SOAM_HOME/5.1/samples/CPP/SharingData`:

   ```
   make
   ```

Package and deploy the sample service

On Windows

To deploy the service, you first need to package it.

1. Go to the directory in which the compiled samples are located:

   ```
   cd %SOAM_HOME%\5.1\samples\CPP\SharingData\Output\
   ```
2. Create the service package by compressing the service executable into a zip file:

   ```
   gzip DataService.exe
   ```

   You have now created your service package `DataService.exe.gz`.

On Linux

To deploy the service, you first need to package it.

1. Change to the directory in which the compiled samples are located:

   ```
   cd $SOAM_HOME/5.1/samples/CPP/SharingData/Output/
   ```
2. Create the service package by compressing the service executable into a tar file:

   ```
   tar -cvf DataService.tar DataService
   ```

   ```
   gzip DataService.tar
   ```

   You have now created your service package `DataService.tar.gz`.

Add the application

When you add an application through the DE PMC, you must use the Add Application wizard. This wizard defines a consumer location to associate with your application, deploys your service package, and registers your application. After completing the steps with the wizard, your application should be ready to use.

1. In the DE PMC, click Symphony Workload > Configure Applications.

   The Applications page displays.
2. Select Global Actions > Add/Remove Applications.

   The Add/Remove Application page displays.
3. Select Add an application, then click Continue.

The Adding an Application page displays.

4. Select Use existing profile and add application wizard. Click Browse and navigate to your application profile.
5. Select your application profile xml file, then click Continue

   For SampleApp, you can find your profile in the following location:

   - C++:
     - Windows—`%SOAM_HOME%\5.1\samples\CPP\SharingData\SharingData.xml`
     - Linux—`$SOAM_HOME/5.1/samples/CPP/SharingData/SharingData.xml`

   The Service Package location window displays.
6. Browse to the service package you created in .gz or tar.gz format and select it, then, click Continue.

   The Confirmation window displays.
7. Review your selections, then click Confirm.

   The window displays indicating progress. Your application is ready to use.
8. Click Close.

   The window closes and you are now back in the Platform Management Console. Your new application is displayed as enabled.

Run the sample client and service

On Windows

To run the service, you run the client application. The service a client application uses is specified in the application profile.

1. Run the client application:

   %SOAM_HOME%\5.1\samples\CPP\SharingData\Output\DataClient.exe

   You should see output on the command line as workload is submitted to the system.

   The client starts and the system starts the corresponding service. The client displays messages indicating that it is running.

On Linux

To run the service, you run the client application. The service a client application uses is specified in the application profile.

1. Run the client application:

   $SOAM_HOME/5.1/samples/CPP/SharingData/Output/DataClient

   You should see output on the command line as workload is submitted to the system.

   The client starts and the system starts the corresponding service. The client displays messages indicating that it is running.

Walk through the code

Locate the code samples

Operating System	Files	Location of Code Sample
Windows	Client	`%SOAM_HOME%\5.1\samples\CPP\SharingData\Client`
	Input, output and data object declaration and implementation	`%SOAM_HOME%\5.1\samples\CPP\SharingData\Common`
	Service code	`%SOAM_HOME%\5.1\samples\CPP\SharingData\Service`
	Application profile	The service required to compute the input data along with additional application parameters are defined in the application profile: `%SOAM_HOME%\5.1\samples\CPP\SharingData \SharingData.xml`
	Output directory	`%SOAM_HOME%\5.1\samples\CPP\SharingData\Output\`
Linux	Client	`$SOAM_HOME/5.1/samples/CPP/SharingData/Client`
	Input, output and data object declaration and implementation	`$SOAM_HOME/5.1/samples/CPP/SharingData/Common`
	Service code	`$SOAM_HOME/5.1/samples/CPP/SharingData/Service`
	Application profile	The service required to compute the input data along with additional application parameters are defined in the application profile: `$SOAM_HOME/5.1/samples/CPP/SharingData/ SharingData.xml`
	Output directory	`$SOAM_HOME/5.1/samples/CPP/SharingData/Output/`

What the client sample does

In the samples, the output message is different from the input message object.

The client creates a session with common data. It sends 10 input messages, and retrieves the output. The client then outputs Hello Grid!!

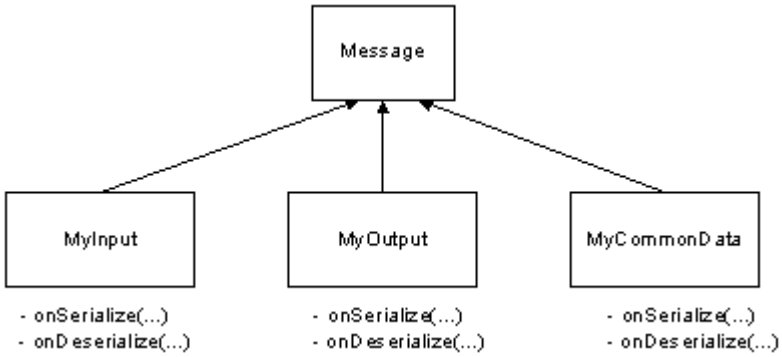
What the service sample does

The service takes input data sent by client applications, returns the input data you have sent and replies "Hello Client !!". The service uses onSessionEnter() to define attributes global to the session.

Prepare common data in your client

1. Declare and implement the Message object.

   In the synchronous client tutorial, input and output message objects were the same object. In this tutorial, different objects represent input and output. In addition, we are creating an additional object to represent common data.



2. Once your message and data objects are declared, implement handlers for serialization and deserialization.

   In `MyDataObjects.cpp`, we implement methods to handle the data.

3. Use the common data object when creating a session:

   a) As in the synchronous client tutorial, initialize the client and connect to the application. Then, create your session to group tasks.

   b) When creating a session, use the common data object to pass data from the client application to the service.

   In `Client.cpp`, we create a session and pass the common data object.

```
...
// Set up session creation attributes
        SessionCreationAttributes attributes;
        attributes.setSessionName("mySession");
        attributes.setSessionType("ShortRunningTasks");
        attributes.setSessionFlags(Session::ReceiveSync);
        attributes.setCommonData(&commonData);
        // Create a synchronous session
                    SessionPtr sesPtr = conPtr->createSession(attributes);
...
```

4. Now proceed the same way as in the synchronous client tutorial:

   - Send input data to be processed
   - Retrieve output
   - Catch exceptions
   - Uninitialize

Access common data in your service

1. Define a service container and get data from the session:

   a) As in the basic service tutorial, first define a service container.

   b) Retrieve the common data from the session sent by the client by implementing onSessionEnter() before your invoke call.

   onSessionEnter() is called once for the duration of the sessions corresponding pair is onSessionLeave().

   In `SampleService.cpp`, we inherited from the ServiceContainer class, and implemented onSessionEnter() to get common data and store it for later with the session context.

```
...
class MyServiceContainer : public ServiceContainer
{
public:
    void onSessionEnter (SessionContextPtr& sessionContext)
    {
        // get the current session ID (if needed)
        m_currentSID = const_cast<char*>(sessionContext->getSessionId());

        // populate our common data object
        m_commonData = new MyCommonData();
        sessionContext->populateCommonData(*m_commonData);
    }
    void onInvoke (TaskContextPtr& taskContext)
{
....
    void onSessionLeave()
    {
        // We get a chance to free the common data here
        if (SOAM_NULL_PTR != m_commonData)
        {
            delete m_commonData;
            m_commonData =  SOAM_NULL_PTR;
        }
    }
...
```

2. Process the input.

   In this example, we use the common data in our invoke call by formatting the output string. We then set our output message as usual to send common data back with each of the replies.

```
...
void onInvoke (TaskContextPtr& taskContext)
    {
 ...
// setup a reply to the client
        std::string str="Client sent : ";
        str += inMsg.getString();
        str += "\nSymphony replied : Hello Client !! with common data (\"";
        str += m_commonData->getString();
        str += "\") for session(";
        str += m_currentSID;
        str += ")";
        outMsg.setString(str.c_str());
        // set our output message
        taskContext->setTaskOutput(outMsg);
    }
...
```

3. Perform any data cleanup: after processing the input, use the onSessionLeave() call to free the data for the session.

   The call onSessionLeave() is called once for every session that is created.

```
...
void onSessionLeave()
    {
        // We get a chance to free the common data here
        if (SOAM_NULL_PTR != m_commonData)
        {
            delete m_commonData;
            m_commonData =  SOAM_NULL_PTR;
        }
    }
...
```

4. As with the basic service, run the container in the service main and catch exceptions.

Java Tutorials

Tutorial: SampleApp: Your first synchronous Symphony Java client

Goal

This tutorial guides you through the process of building, packaging, deploying, and running the sample client and service. It then walks you through the sample client application code.

You will learn the minimum amount of code that you need to create a synchronous client.

At a glance

1. Build the sample client and service
2. Package the sample service
3. Add the application
4. Run the sample client and service
5. Walk through the code

Build the sample client and service

On Windows

Compile with the .bat file

You can build client application and service samples at the same time.

1. Change to the `%SOAM_HOME%\5.1\samples\Java\SampleApp` directory and run the .bat file:

   ```
   build.bat
   ```

Compile with the Ant build file

You can build client application and service samples at the same time.

1. Change to the `%SOAM_HOME%\5.1\samples\Java\SampleApp` directory and run the command:

   ```
   ant
   ```

Compile in Eclipse

To compile in Eclipse, see "Symphony plug-in for Eclipse" in the *Application Development Guide.*

On Linux

You can build client application and service samples at the same time.

1. Change to the conf directory under the directory in which you installed Symphony DE.
2. Set the environment:

- For csh, enter

  ```
  source cshrc.soam
  ```

- For bash, enter

  ```
  . profile.soam
  ```

3. Compile with the Makefile or with the Ant build file.

   - Compile with the Makefile:

     Change to the `$SOAM_HOME/5.1/samples/Java/SampleApp` directory and run the make command:

     ```
     make
     ```

   - Compile with the Ant build file:

     Change to the `$SOAM_HOME/5.1/samples/Java/SampleApp` directory and run the build command:

     ```
     ant
     ```

Package the sample service

You must package the files required by your service to create a service package. When you built the sample, the service package was automatically created for you.

1. Your service package SampleServiceJavaPackage.jar is in the following directory:

   ```
   %SOAM_HOME%\5.1\samples\Java\SampleApp
   ```

Add the application

When you add an application through the DE PMC, you must use the Add Application wizard. This wizard defines a consumer location to associate with your application, deploys your service package, and registers your application. After completing the steps with the wizard, your application should be ready to use.

1. In the DE PMC, click Symphony Workload > Configure Applications.

   The Applications page displays.

2. Select Global Actions > Add/Remove Applications.

   The Add/Remove Application page displays.

3. Select Add an application, then click Continue.

   The Adding an Application page displays.

4. Select Use existing profile and add application wizard. Click Browse and navigate to your application profile.

5. Select your application profile xml file, then click Continue

   For SampleApp, you can find your profile in the following location:

   - Java
     - Windows—`%SOAM_HOME%\5.1\samples\Java\SampleApp\SampleAppJava.xml`
     - Linux—`$SOAM_HOME/5.1/samples/Java/SampleApp/SampleAppJava.xml`

   The Service Package location window displays.

6. Browse to the created service package and select it, then, click Continue.

- Java
  - Windows—`%SOAM_HOME%\5.1\samples\Java\SampleApp\SampleServiceJavaPackage.jar`
  - Linux—`$SOAM_HOME/5.1/samples/Java/SampleApp/SampleServiceJavaPackage.zip`

The Confirmation window displays.

7. Review your selections, then click Confirm.

   The window displays indicating progress. Your application is ready to use.

8. Click Close.

   The window closes and you are now back in the Platform Management Console. Your new application is displayed as enabled.

Run the sample client and service

On Windows

To run the service, you run the client application. The service a client application uses is specified in the application profile.

1. Run the client application.

   - From the command-line:

   %SOAM_HOME%\5.1\samples\Java\SampleApp\RunSyncClient.bat

You should see output as work is submitted to the system.

The client starts and the system starts the corresponding service. The client displays messages indicating that it is running.

On Linux

To run the service, you run the client application. The service a client application uses is specified in the application profile.

1. Run the client application:

   $SOAM_HOME/5.1/samples/Java/SampleApp/RunSyncClient.sh

You should see output as work is submitted to the system.

The client starts and the system starts the corresponding service. The client displays messages indicating that it is running.

Walk through the code

Review the sample client application code to learn how you can create a synchronous client application.

Locate the code samples

Operating System	Files	Location of Code Sample
Windows	Client	`%SOAM_HOME%\5.1\samples\Java\SampleApp\src\com\platform\symphony\samples\SampleApp\client\SyncClient.java`
	Input and output objects	`%SOAM_HOME%\5.1\samples\Java\SampleApp\src\com\platform\symphony\samples\SampleApp\common\MyInput.java` `%SOAM_HOME%\5.1\samples\Java\SampleApp\src\com\platform\symphony\samples\SampleApp\common\MyOutput.java`
	Service code	`%SOAM_HOME%\5.1\samples\Java\SampleApp\src\com\platform\symphony\samples\SampleApp\service\MyService.java`
	Application profile	The service required to compute the input data along with additional application parameters are defined in the application profile: `%SOAM_HOME%\5.1\samples\Java\SampleApp\SampleAppJava.xml`
	Output directory	`%SOAM_HOME%\5.1\samples\Java\SampleApp\`
Linux	Client	`$SOAM_HOME/5.1/samples/Java/SampleApp/src/com/platform/symphony/samples/SampleApp/client/syncClient.java`
	Input and output objects	`$SOAM_HOME/5.1/samples/Java/SampleApp/src/com/platform/symphony/samples/SampleApp/common/MyInput.java` `$SOAM_HOME/5.1/samples/Java/SampleApp/src/com/platform/symphony/samples/SampleApp/common/MyOutput.java`
	Service code	`$SOAM_HOME/5.1/samples/Java/SampleApp/src/com/platform/symphony/samples/SampleApp/service/MyService.java`
	Application profile	The service required to compute the input data along with additional application parameters are defined in the application profile: `$SOAM_HOME/5.1/samples/Java/SampleApp/SampleAppJava.xml`
	Output directory	`$SOAM_HOME/5.1/samples/Java/SampleApp/`

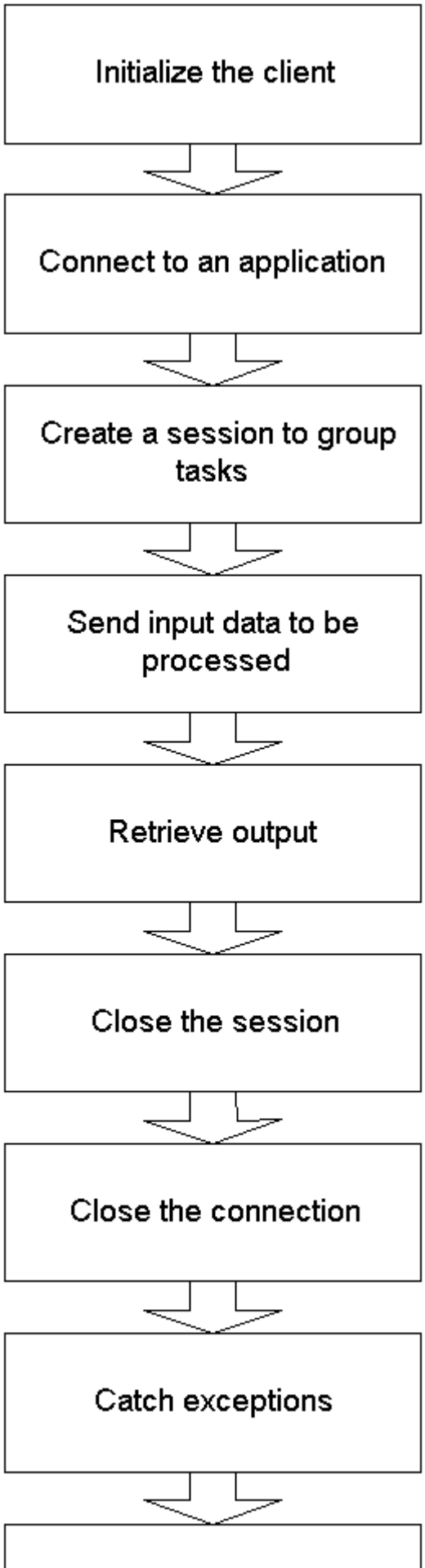
What the sample does

The client application sample sends 10 input messages through Symphony to the service with the data “Hello Grid!!”. The client blocks to receive messages synchronously.

The service takes the input data sent by the client and returns it with the additional data “Hello Client !!”.

Synchronous client structure

Before developing your client code, implement input and output objects that implement java.io.Serializable. Then, create your Symphony client.


Initialize the client
Connect to an application
Create a session to group tasks
Send input data to be processed
Retrieve output
Close the session
Close the connection
Catch exceptions

Implement input and output objects

Implement the MyInput class

The myInput class acts as input to the service. In `MyInput.java`, we implement methods to set and access the data, such as the message string and task ID.

The input object must implement java.io.Serializable. Making the object serializable means that Java knows how to deconstruct the object so that it can be passed through the network to the service. This also means that Java knows how to reconstruct the object when it is received by the service.

```
...
public class MyInput implements Serializable
{
    //============================================================================
    //  Constructors
    //============================================================================
    public MyInput()
    {
        super();
        m_id = 0;
    }
    public MyInput(int id, String string)
    {
        super();
        m_id = id;
        m_string = string;
    }
    //============================================================================
    //  Accessors and Mutators
    //============================================================================

    public int getId()
    {
        return m_id;
    }
    public void setId(int id)
    {
        m_id = id;
    }
    public String getString()
    {
        return m_string;
    }
    public void setString(String string)
    {
        m_string = string;
    }
...
```

Implement the MyOutput class

The myoutput object is the result of the computation of input to the service, and is returned to the client by the service.

In `MyOutput.java`, we implement methods to set and access the output data, such as the message string, task ID, and run time that is returned from the service. Similar to the input object, the output object must implement java.io.Serializable.

```
...
public class MyOutput implements Serializable{
    //============================================================================
    //  Constructor
    //============================================================================
    public MyOutput()
    {
        super();
        m_id = 0;
    }
    //============================================================================
    //  Accessors and Mutators
    //============================================================================
    public int getId()
    {
        return m_id;
    }
    public void setId(int id)
    {
        m_id = id;
    }
    public String getRunTime()
    {
        return m_runTime;
    }
    public void setRunTime(String runTime)
    {
        m_runTime = runTime;
    }
    public String getString()
    {
        return m_string;
    }

    public void setString(String string)
    {
        m_string = string;
    }
...
```

Initialize the client

In `SyncClient.java`, when you initialize, you initialize the Symphony client infrastructure. You initialize once per client.

Important:

Initialization is required. Otherwise, API calls fail.

```
...
SoamFactory.initialize();
...
```

Connect to an application

A connection establishes a context for your client and workload. When you connect to an application:

- Application attributes defined in the application profile are used to provide context such as which service to use, session type, and any additional scheduling or application parameters.
- A connection object is returned.

The application name in the connection must match that defined in the application profile.

The default security callback encapsulates the callback for the user name and password. In Symphony DE, there is no security checking and login credentials are ignored —you can specify any user name and

password. However, when using your client on the grid with Platform Symphony, you need a valid user name and password.

```
...
// Set up application specific information to be supplied to Symphony
            String appName="SampleAppJava";
           // Set up application authentication information using the default security provider
           DefaultSecurityCallback securityCB = new DefaultSecurityCallback("Guest", "Guest");
            Connection connection = null;
            try
            {
                // Connect to the specified application
                connection = SoamFactory.connect(appName, securityCB);
                // Retrieve and print our connection ID
                System.out.println("connection ID=" + connection.getId());
...
```

```
finally
            {
                // Mandatory connection close
                if (connection != null)
                {
                    connection.close();
                    System.out.println("Connection closed");
                }
                                }
...
```

Important:

The creation and usage of the connection object must be scoped in a try-finally block. The finally block, with the connection.close() method, ensures that the connection is always closed whether exceptional behavior occurs or not. Failure to close the connection causes the connection to continue to occupy system resources.

Create a session to group tasks

A session is a way of logically grouping tasks that are sent to a service for execution. The tasks are sent and received synchronously.

```
...
 // Set up session attributes
                SessionCreationAttributes attributes = new SessionCreationAttributes();
                attributes.setSessionName("mySession");
                attributes.setSessionType("ShortRunningTasks");
                attributes.setSessionFlags(Session.SYNC);
                // Create a synchronous session
                Session session = null;
                try
                {
                    session = connection.createSession(attributes);
...
                finally
                {
                    // Mandatory session close
                    if (session != null)
                    {
                        session.close();
                        System.out.println("Session closed");
                    }
                }
...
```

When creating a synchronous session, you need to specify the session attributes by using the SessionCreationAttributes object. In this sample, we create a SessionCreationAttributes object called attributes and set three parameters in the object.

The first parameter is the session name. This is optional. The session name can be any descriptive name you want to assign to your session. It is for informational purposes, such as in the command line interface.

The second parameter is the session type. The session type is optional. You can leave this parameter blank and system default values are used for your session.

The third parameter is the session flag, which we specify as Session.SYNC. This indicates to Symphony that this is a synchronous session.

We pass the attributes object to the createSession() method, which returns the created session.

Important:

Similar to the connection object, the creation and usage of the session (sending and receiving data) must be scoped in a try-finally block. The finally block, with the session.close() method, ensures that the session is always closed, whether exceptional behavior occurs or not. Failure to close the session causes the session to continue to occupy system resources.

Send input data to be processed

In this step, we create 10 input messages to be processed by the service. When a message is sent, a task input handle is returned. This task input handle contains the ID for the task that was created for this input message.

```
...
// Now we will send some messages to our service
                    int tasksToSend = 10;
                    for (int taskCount = 0; taskCount < tasksToSend; taskCount++)
                    {
                        // Create a message
                        MyInput myInput = new MyInput(taskCount, "Hello Grid !!");
                        // Set task submission attributes
                        TaskSubmissionAttributes taskAttr = new TaskSubmissionAttributes();
                        taskAttr.setTaskInput(myInput);
                        // Send it
                        TaskInputHandle input = session.sendTaskInput(taskAttr);
                        // Retrieve and print task ID
                        System.out.println("task submitted with ID : " + input.getId());
                                                }
...
```

Retrieve output

The call fetchTaskOutput() blocks until the output for all tasks is retrieved. If there is output to retrieve, getTaskOutput() gets the output

Important:

Results are not sent back in order. If order of results is important, the client application must sort the results.

```
...
// Now get our results - will block here until all tasks retrieved
                    EnumItems enumOutput = session.fetchTaskOutput(tasksToSend);
                    // Inspect results
                    TaskOutputHandle output = enumOutput.getNext();
                    while (output != null)
                    {
                        // Check for success of task
                        if (output.isSuccessful())
                        {
                            // Get the message returned from the service
                            MyOutput myOutput = (MyOutput)output.getTaskOutput();
                            // Display content of reply
                            System.out.println("\nTask Succeeded [" +  output.getId() + "]");
                            System.out.println("Your Internal ID was : " + myOutput.getId());
                      System.out.println("Estimated runtime was recorded as : " +
myOutput.getRunTime());
                      System.out.println(myOutput.getString());
                        }
                        else
                        {
                            // Get the exception associated with this task
                            SoamException ex = output.getException();
                            System.out.println("Task Failed : ");
                            System.out.println(ex.toString());
                        }
                        output = enumOutput.getNext();
                                                                 }
 ...
```

Catch exceptions

Any exceptions thrown take the form of SoamException. Catch all Symphony exceptions that occurred in the client application, service, and system.

The sample code in Retrieve output catches exceptions of type SoamException.

Uninitialize

Always uninitialize the client at the end of all API calls. If you do not call uninitialize, the client stays in an undefined state and resources used by the client are held indefinitely.

Important:

Once you uninitialize, all objects become invalid. For example, you can no longer create a session or send an input message.

```
...
SoamFactory.uninitialize();
...
```

Tutorial: SampleApp: Developing an asynchronous Symphony Java client

Goal

In this tutorial, you will learn how to convert a synchronous client into an asynchronous one.

At a glance

1. Build the sample client and service
2. Package the sample service
3. Add the application
4. Run the sample client and service
5. Walk through the code

Build the sample client and service

On Windows

Compile with the .bat file

You can build client application and service samples at the same time.

1. Change to the `%SOAM_HOME%\5.1\samples\Java\SampleApp` directory and run the .bat file:

   ```
   build.bat
   ```

Compile with the Ant build file

You can build client application and service samples at the same time.

1. Change to the `%SOAM_HOME%\5.1\samples\Java\SampleApp` directory and run the command:

   ```
   ant
   ```

Compile in Eclipse

To compile in Eclipse, see "Symphony plug-in for Eclipse" in the *Application Development Guide.*

On Linux

You can build client application and service samples at the same time.

1. Change to the conf directory under the directory in which you installed Symphony DE.
2. Set the environment:
   - For csh, enter

     ```
     source cshrc.soam
     ```
   - For bash, enter

     ```
     . profile.soam
     ```
3. Compile with the Makefile or with the Ant build file.
   - Compile with the Makefile:

     Change to the `$SOAM_HOME/5.1/samples/Java/SampleApp` directory and run the make command:

     ```
     make
     ```
   - Compile with the Ant build file:

Change to the `$SOAM_HOME/5.1/samples/Java/SampleApp` directory and run the build command:

```
ant
```

Package the sample service

You must package the files required by your service to create a service package. When you built the sample, the service package was automatically created for you.

1. Your service package SampleServiceJavaPackage.jar is in the following directory:

   ```
   %SOAM_HOME%\5.1\samples\Java\SampleApp
   ```

Add the application

When you add an application through the DE PMC, you must use the Add Application wizard. This wizard creates a consumer location to associate with your application, deploys your service package, and registers your application. After completing the steps with the wizard, your application should be ready to use.

1. In the DE PMC, click Symphony Workload > Configure Applications.

   The Applications page displays.
2. Select Global Actions > Add/Remove Applications.

   The Add/Remove Application page displays.
3. Select Add an application, then click Continue.

   The Adding an Application page displays.
4. Select Use existing profile and add application wizard. Click Browse and navigate to your application profile.
5. Select your application profile xml file, then click Continue

   For SampleApp, you can find your profile in the following location:

   - Java
     - Windows—`%SOAM_HOME%\5.1\samples\Java\SampleApp\SampleAppJava.xml`
     - Linux—`$SOAM_HOME/5.1/samples/Java/SampleApp/SampleAppJava.xml`

   The Service Package location window displays.
6. Browse to the created service package and select it, then, select Continue.

   - Java
     - Windows—`%SOAM_HOME%\5.1\samples\Java\SampleApp \SampleServiceJavaPackage.jar`
     - Linux—`$SOAM_HOME/5.1/samples/Java/SampleApp/ SampleServiceJavaPackage.zip`

   The Confirmation window displays.
7. Review your selections, then click Confirm.

   The window displays indicating progress. Your application is ready to use.
8. Click Close.

The window closes and you are now back in the Platform Management Console. Your new application is displayed as enabled.

Run the sample client and service

On Windows

To run the service, you run the client application. The service a client application uses is specified in the application profile.

1. Run the client application.
   - From the command line:

   %SOAM_HOME%\5.1\samples\Java\SampleaApp\RunAsyncClient.bat

You should see output as work is submitted to the system.

The client starts and the system starts the corresponding service. The client displays messages indicating that it is running.

On Linux

To run the service, you run the client application. The service a client application uses is specified in the application profile.

1. Run the client application:

   $SOAM_HOME/5.1/samples/Java/SampleApp/RunAsyncClient.sh

You should see output on the command line as work is submitted to the system.

The client starts and the system starts the corresponding service. The client displays messages indicating that it is running.

Walk through the code

Review the sample client application code to learn how you can understand the differences between a synchronous client and an asynchronous client.

Locate the code samples

Operating System	Files	Location of Code Sample
Windows	Client	`%SOAM_HOME%\5.1\samples\Java\SampleApp\src\com\platform\symphony\samples\SampleApp\client\AsyncClient.java`
	Input and output objects	`%SOAM_HOME%\5.1\samples\Java\SampleApp\src\com\platform\symphony\samples\SampleApp\common\MyInput.java` `%SOAM_HOME%\5.1\samples\Java\SampleApp\src\com\platform\symphony\samples\SampleApp\common\MyOutput.java`
	Service code	`%SOAM_HOME%\5.1\samples\Java\SampleApp\src\com\platform\symphony\samples\SampleApp\service\MyService.java`
	Application profile	The service required to compute the input data along with additional application parameters are defined in the application profile: `%SOAM_HOME%\5.1\samples\Java\SampleApp\SampleAppJava.xml`
	Output directory	`%SOAM_HOME%\5.1\samples\Java\SampleApp\`
Linux	Client	`$SOAM_HOME/5.1/samples/Java/SampleApp/src/com/platform/symphony/samples/SampleApp/client/AsyncClient.java`
	Input and output objects	`$SOAM_HOME/5.1/samples/Java/SampleApp/src/com/platform/symphony/samples/SampleApp/common/MyInput.java` `$SOAM_HOME/5.1/samples/Java/SampleApp/src/com/platform/symphony/samples/SampleApp/common/MyOutput.java`
	Service code	`$SOAM_HOME/5.1/samples/Java/SampleApp/src/com/platform/symphony/samples/SampleApp/service/MyService.java`
	Application profile	The service required to compute the input data along with additional application parameters are defined in the application profile: `$SOAM_HOME/5.1/samples/Java/SampleApp/SampleAppJava.xml`
	Output directory	`$SOAM_HOME/5.1/samples/Java/SampleApp/`

What the sample does

The client application sample sends 10 input messages through Symphony to the service with the data “Hello Grid !!”.

The service takes the input data sent by the client and returns it with the additional data “Hello Client !!”.

Results are returned asynchronously with a callback interface provided by the client to the API. Methods in this interface are called from threads within the API when certain events occur. In the sample, the events are:

- When there is an error at the session level
- When results return from Symphony

Considerations for asynchronous clients

The purpose of an asynchronous client is to get the output as soon as it is available. The client thread does not need to be blocked once the input data is sent and can perform other actions.

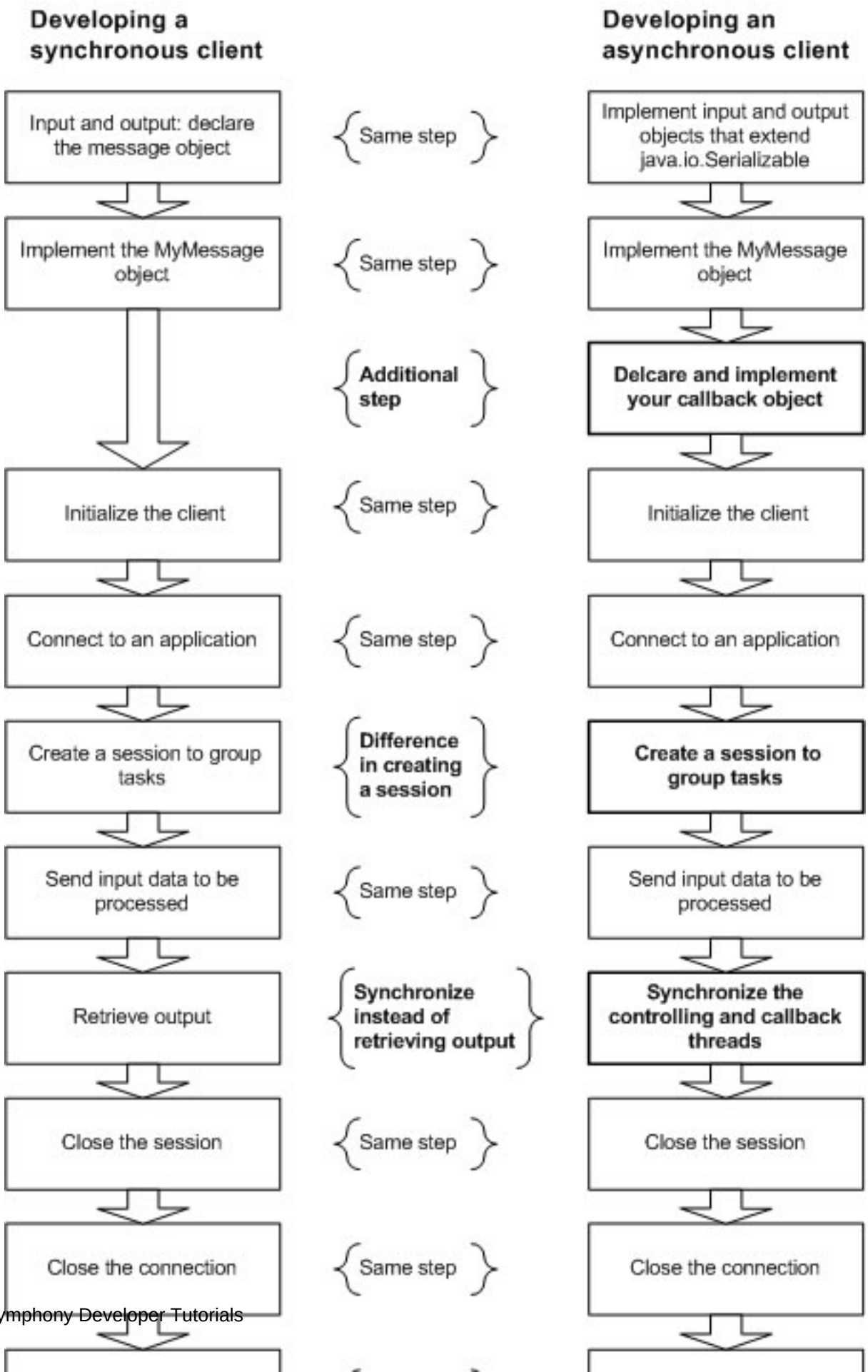
Synchronization	Because results can come back at any time, it is probable that your callback code needs synchronization between the callback thread and the controlling thread. The controlling thread needs to know when work is complete.
Order of results	Results are not sent back in order. If order of results is important, the client application must sort the results.

Code differences between synchronous and asynchronous clients

An asynchronous client is very similar to a synchronous client. The only differences are:

- You need to specify a callback when creating a session
- You specify a different flag to indicate asynchronous when you create a session
- Handling of replies

The following figure highlights in bold the differences between synchronous and asynchronous clients. Everything else is the same as the synchronous client.


Developing a synchronous client
Developing an asynchronous client
Input and output: declare the message object
Same step
Implement input and output objects that extend java.io.Serializable
Implement the MyMessage object
Same step
Implement the MyMessage object
Additional step
Delcare and implement your callback object
Initialize the client
Same step
Initialize the client
Connect to an application
Same step
Connect to an application
Create a session to group tasks
Difference in creating a session
Create a session to group tasks
Send input data to be processed
Same step
Send input data to be processed
Retrieve output
Synchronize instead of retrieving output
Synchronize the controlling and callback threads
Close the session
Same step
Close the session
Close the connection
Same step
Close the connection

Implement input and output objects and initialize the client

As in the synchronous client tutorial, implement your own input and output objects and initialize the client.

For more details, look at the synchronous client tutorial, specifically:

- Implement input and output objects
- Initialize the client

Declare and implement your callback object

Perform this step after implementing your own input and output objects.

In `MySessionCallback.java`, we create our own callback class that extends the SessionCallback class, and we implemented onResponse() to retrieve the output for each input message that we send.

Note that:

- We handle when an exception occurs on the callback method for the session. If you do not handle the exception, you do not have any exceptions if an error occurs on the callback.
- onResponse() is called every time a task completes and output is returned to the client. The task output handle allows the client code to manipulate the output.
- isSuccessful() checks whether there is output to retrieve.
- If there is output to retrieve, getTaskOutput() gets the output. Once results return, we print them to standard output and return.

```
public class MySessionCallback extends SessionCallback
{
    //========================================================================
    //  Constructor
    //========================================================================
    public MySessionCallback(int tasksToReceive)
    {
        m_tasksReceived = 0;
        m_exception = false;
        m_tasksToReceive = tasksToReceive;
}
```

```
    //========================================================================
    //  SessionCallback Interface Methods
    //========================================================================

    /**
     * Invoked when an exception occurs within the scope of the given session.
     * Must be implemented by the application developer.
     */
    public void onException(SoamException exception) throws SoamException
    {
        System.out.println("An exception occured on the callback :");
        System.out.println(exception.getMessage());
        setException(true);
    }
```

```
/**
     * Invoked when a task response is ready.
     * Must be implemented by the application developer.
     */
    public void onResponse(TaskOutputHandle output) throws SoamException
    {
        try
        {
            // check for success of task
            if (output.isSuccessful())
            {
                // get the message returned from the service
                MyOutput myOutput = (MyOutput)output.getTaskOutput();
                // display content of reply
                System.out.println("\nTask Succeeded [" +  output.getId() + "]");
                System.out.println("Your Internal ID was : " + myOutput.getId());
               System.out.println("Estimated runtime was recorded as : " + myOutput.getRunTime
());
                System.out.println(myOutput.getString());
            }
            else
            {
                // get the exception associated with this task
                SoamException ex = output.getException();
                System.out.println("Task Failed :");
                System.out.println(ex.getMessage());
            }
        }
        catch (Exception exception)
        {
            System.out.println("Exception occured in onResponse() : ");
            System.out.println(exception.getMessage());
        }

        // Update counter used to synchronize the controlling thread
        // with this callback object
        incrementTaskCount();
        }
```

Create a session to group tasks

In `AsyncClient.java`, perform this step after you have connected to the application.

When creating an asynchronous session:

- Specify the flag PARTIAL_ASYNC. This indicates that results are collected asynchronously.
- Provide a callback object.

```
...
// Create session callback
                int tasksToSend = 10;
                MySessionCallback myCallback = new MySessionCallback(tasksToSend);
                Session session = null;
                // Set up session attributes
                SessionCreationAttributes attributes = new SessionCreationAttributes();
                attributes.setSessionName("mySession");
                attributes.setSessionType("ShortRunningTasks");
                attributes.setSessionFlags(Session.PARTIAL_ASYNC);
                attributes.setSessionCallback(myCallback);
                // Create an asynchronous session
                try
                {
                    session = connection.createSession(attributes);
                    // Retrieve and print session ID
                    System.out.println("Session ID:" + session.getId());
...
```

Synchronize the controlling and callback threads

Perform this step after sending the input data to be processed.

Since our client is asynchronous, we need to synchronize the controlling thread and the callback thread. In this example, the controlling thread blocks until all replies have come back.

The callback signals when all results are received.

```
...
synchronized(myCallback)
                    {
                        while (!myCallback.isDone())
                        {
                            myCallback.wait();
                        }
                    }
...
```

Tutorial: SampleApp: Your first Symphony Java service

Goal

This tutorial walks you through the sample service code, then guides you through the process of building and running a service.

You learn the minimum amount of code that you need to create a service.

At a glance

1. Build the sample client and service
2. Package and deploy the sample service
3. Run the sample client and service
4. Walk through the code

Build the sample client and service

On Windows

Compile with the .bat file

You can build client application and service samples at the same time.

1. Change to the `%SOAM_HOME%\5.1\samples\Java\SampleApp` directory and run the .bat file:

   build.bat

Compile with the Ant build file

You can build client application and service samples at the same time.

1. Change to the `%SOAM_HOME%\5.1\samples\Java\SampleApp` directory and run the command:

   ```
   ant
   ```

Compile in Eclipse

To compile in Eclipse, see "Symphony plug-in for Eclipse" in the *Application Development Guide.*

On Linux

You can build client application and service samples at the same time.

1. Change to the conf directory under the directory in which you installed Symphony DE.
2. Set the environment:
   - For csh, enter

     ```
     source cshrc.soam
     ```
   - For bash, enter

     ```
     . profile.soam
     ```
3. Compile with the Makefile or with the Ant build file.
   - Compile with the Makefile:

     Change to the `$SOAM_HOME/5.1/samples/Java/SampleApp` directory and run the make command:

     ```
     make
     ```
   - Compile with the Ant build file:

     Change to the `$SOAM_HOME/5.1/samples/Java/SampleApp` directory and run the build command:

     ```
     ant
     ```

Package the sample service

You must package the files required by your service to create a service package. When you built the sample, the service package was automatically created for you.

1. Your service package SampleServiceJavaPackage.jar is in the following directory:

   ```
   %SOAM_HOME%\5.1\samples\Java\SampleApp
   ```

Add the application

When you add an application through the DE PMC, you must use the Add Application wizard. This wizard defines a consumer location to associate with your application, deploys your service package, and registers your application. After completing the steps with the wizard, your application should be ready to use.

1. In the De PMC, click Symphony Workload > Configure Applications.

   The Applications page displays.
2. Select Global Actions > Add/Remove Applications.

   The Add/Remove Application page displays.
3. Select Add an application, then click Continue.

The Adding an Application page displays.

4. Select Use existing profile and add application wizard. Click Browse and navigate to your application profile.
5. Select your application profile xml file, then click Continue

   For SampleApp, you can find your profile in the following location:

   - Java
     - Windows—`%SOAM_HOME%\5.1\samples\Java\SampleApp\SampleAppJava.xml`
     - Linux—`$SOAM_HOME/5.1/samples/Java/SampleApp/SampleAppJava.xml`

   The Service Package location window displays.
6. Browse to the created service package and select it, then, select Continue.

   - Java
     - Windows—`%SOAM_HOME%\5.1\samples\Java\SampleApp\SampleServiceJavaPackage.jar`
     - Linux—`$SOAM_HOME/5.1/samples/Java/SampleApp/SampleServiceJavaPackage.zip`

   The Confirmation window displays.
7. Review your selections, then click Confirm.

   The window displays indicating progress. Your application is ready to use.
8. Click Close.

   The window closes and you are now back in the Platform Management Console. Your new application is displayed as enabled.

Run the sample client and service

On Windows

To run the service, you run the client application. The service that a client application uses is specified in the application profile.

1. Run the client application:

   - From the command-line:

   %SOAM_HOME%\5.1\samples\Java\SampleApp\RunSyncClient.bat

You should see output as work is submitted to the system.

The client starts and the system starts the corresponding service. The client displays messages indicating that it is running.

On Linux

To run the service, you run the client application. The service that a client application uses is specified in the application profile.

1. Run the client application:

   $SOAM_HOME/5.1/samples/Java/SampleApp/RunSyncClient.sh

You should see output on the command line as work is submitted to the system.

The client starts and the system starts the corresponding service. The client displays messages indicating that it is running.

Walk through the code

Review the sample service code to learn how you can create a service.

Locate the code samples

Operating System	Files	Location of Code Sample
Windows	Client	`%SOAM_HOME%\5.1\samples\Java\SampleApp\src\com\platform\symphony\samples\SampleApp\client\SyncClient.java`
	Input and output objects	`%SOAM_HOME%\5.1\samples\Java\SampleApp\src\com\platform\symphony\samples\SampleApp\common\MyInput.java` `%SOAM_HOME%\5.1\samples\Java\SampleApp\src\com\platform\symphony\samples\SampleApp\common\MyOutput.java`
	Service code	`%SOAM_HOME%\5.1\samples\Java\SampleApp\src\com\platform\symphony\samples\SampleApp\service\MyService.java`
	Application profile	The service required to compute the input data along with additional application parameters are defined in the application profile: `%SOAM_HOME%\5.1\samples\Java\SampleApp\SampleAppJava.xml`
	Output directory	`%SOAM_HOME%\5.1\samples\Java\SampleApp\`
Linux	Client	`$SOAM_HOME/5.1/samples/Java/SampleApp/src/com/platform/symphony/samples/SampleApp/client/SyncClient.java`
	Input and output objects	`$SOAM_HOME/5.1/samples/Java/SampleApp/src/com/platform/symphony/samples/SampleApp/common/MyInput.java` `$SOAM_HOME/5.1/samples/Java/SampleApp/src/com/platform/symphony/samples/SampleApp/common/MyOutput.java`
	Service code	`$SOAM_HOME/5.1/samples/Java/SampleApp/src/com/platform/symphony/samples/SampleApp/service/MyService.java`
	Application profile	The service required to compute the input data along with additional application parameters are defined in the application profile: `$SOAM_HOME/5.1/samples/Java/SampleApp/SampleAppJava.xml`

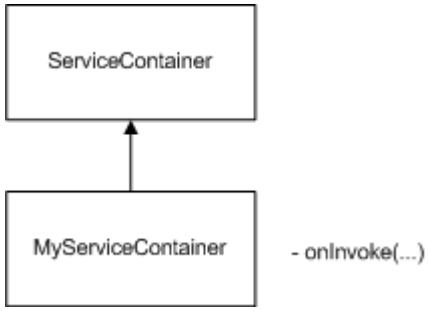
Operating System	Files	Location of Code Sample
	Output directory	`$SOAM_HOME/5.1/samples/Java/SampleApp/`

What the sample does

The service takes input data sent by the client and returns the input data with the additional "Hello Client !!".

Define a service container

For a service to be managed by Symphony, it needs to be in a container object. This is the service container.



In `MyService.java`, MyService inherits (extends) the ServiceContainer class. Once you inherit from the ServiceContainer class, implement handlers so that the service can function properly.

```
...
public class MyService extends ServiceContainer
...
```

Process the input

Symphony calls onInvoke() on the service container once per task. This is where the calculation is performed.

To gain access to the data set from the client, you call the getTaskInput() method on the taskContext. The middleware is responsible for placing the input into the taskContext object.

The task context contains all information and functionality that is available to the service during an onInvoke() call in relation to the task that is being processed.

Important:

Services are virtualized. As a result, a service should not read from stdin or write to stdout. Services can, however, read from and write to files that are accessible to all compute hosts.

```
...
public void onInvoke (TaskContext taskContext) throws SoamException
    {
        // We simply echo the data back to the client
        MyOutput myOutput = new MyOutput();
        // estimate and set our runtime
        Date date = new Date();
        myOutput.setRunTime(date.toString());
        // get the input that was sent from the client
        MyInput myInput = (MyInput)taskContext.getTaskInput();
        // echo the ID
        myOutput.setId(myInput.getId());
        // setup a reply to the client
        StringBuffer sb = new StringBuffer();
        sb.append("Client sent : ");
        sb.append(myInput.getString());
        sb.append("\nSymphony replied : Hello Client !!");
        myOutput.setString(sb.toString());
        // set our output message
        taskContext.setTaskOutput(myOutput);
         }
...
```

Run the container

Create an instance of the service container and run it within the main function.

```
...
public static void main(String args[])
    {
        // Return value of our service program
        int retVal = 0;
        try
        {
            // Create the service container and run it
            MyService myContainer = new MyService();
            myContainer.run();
        }
...
```

Catch exceptions

Catch exceptions in case the container fails to start running.

```
...
catch (Exception ex)
        {
            // Report exception
            System.out.println("Exception caught:");
            System.out.println(ex.toString());
            retVal = -1;
                     }
...
```

Tutorial: SharingData: Developing a Java client and service to share data among tasks

Goal

This tutorial walks you through how to develop a client application and service to share data among all tasks in a session. The data is shared by all invocations of tasks within the same session.

You learn how to use different data objects for input, output, and common data.

At a glance

1. Build the sample client and service
2. Package the sample service
3. Add the application
4. Run the sample client and service
5. Walk through the code

Build the sample client and service

On Windows

Compile with the .bat file

You can build client application and service samples at the same time.

1. Change to the `%SOAM_HOME%\4.0\samples\Java\SharingData` directory and run the .bat file:

   ```
   build.bat
   ```

Compile with the Ant build file

You can build client application and service samples at the same time.

1. Change to the %SOAM_HOME%\4.0\samples\Java\SharingData directory and run the command:

   ```
   ant
   ```

Compile in Eclipse

To compile in Eclipse, see "Symphony plug-in for Eclipse" in the *Application Development Guide.*

On Linux

You can build client application and service samples at the same time.

1. Change to the conf directory under the directory in which you installed Symphony DE.
2. Set the environment:
   - For csh, enter

     ```
     source cshrc.soam
     ```
   - For bash, enter

     ```
     . profile.soam
     ```
3. Compile with the Makefile or with the Ant build file.
   - Compile with the Makefile:

Change to the `$SOAM_HOME/4.0/samples/Java/SharingData directory` and run the make command:

```
make
```

- Compile with the Ant build file:

  Change to the `$SOAM_HOME/4.0/samples/Java/SharingData` directory and run the build command:

  ```
  ant
  ```

Package the sample service

You must package the files required by your service to create a service package. When you built the sample, the service package was automatically created for you.

1. Your service package SampleServiceJavaPackage.jar is in the following directory:

   ```
   cd %SOAM_HOME%\5.1\samples\Java\SharingData
   ```

Add the application

When you add an application through the DE PMC, you must use the Add Application wizard. This wizard defines a consumer location to associate with your application, deploys your service package, and registers your application. After completing the steps with the wizard, your application should be ready to use.

1. In the DE PMC, click Symphony Workload > Configure Applications.

   The Applications page displays.
2. Select Global Actions > Add/Remove Applications.

   The Add/Remove Application page displays.
3. Select Add an application, then click Continue.

   The Adding an Application page displays.
4. Select Use existing profile and add application wizard. Click Browse and navigate to your application profile.
5. Select your application profile xml file, then click Continue

   For SampleApp, you can find your profile in the following location:

   - Java
     - Windows—`%SOAM_HOME%\5.1\samples\Java\SharingData\SharingDataJava.xml`
     - Linux—`$SOAM_HOME/5.1/samples/Java/SharingData/SharingDataJava.xml`

   The Service Package location window displays.
6. Browse to the created service package and select it, then, select Continue.

   - Java
     - Windows—`%SOAM_HOME%\5.1\samples\Java\SharingData\SharingDataServiceJavaPackage.jar`
     - Linux—`$SOAM_HOME/5.1/samples/Java/SharingData/SharingDataServiceJavaPackage.zip`

The Confirmation window displays.

7. Review your selections, then click Confirm.

   The window displays indicating progress. Your application is ready to use.

8. Click Close.

   The window closes and you are now back in the Platform Management Console. Your new application is displayed as enabled.

Run the sample client and service

On Windows

To run the service, you run the client application. The service that a client application uses is specified in the application profile.

1. Run the client application:

   - From the command-line:

   %SOAM_HOME%\5.1\samples\Java\SharingData\RunSharingDataClient.bat

You should see output as work is submitted to the system.

The client starts and the system starts the corresponding service. The client displays messages indicating that it is running.

On Linux

To run the service, you run the client application. The service a client application uses is specified in the application profile.

1. Run the client application:

   $SOAM_HOME/5.1/samples/Java/SharingData/RunSharingDataClient.sh

You should see output as work is submitted to the system.

The client starts and the system starts the corresponding service. The client displays messages indicating that it is running.

Walk through the code

Review the sample code to learn how you can create a client and service that uses data.

Locate the code samples

Operating System	Files	Location of Code Sample
Windows	Client	`%SOAM_HOME%\5.1\samples\Java\SharingData\src\com\platform\symphony\samples\SharingData\client\SharingDataClient.java`

	Input, output, and data objects	`%SOAM_HOME%\5.1\samples\Java\SharingData\src\com\platform\symphony\samples\SharingData\common\MyInput.java`
		`%SOAM_HOME%\5.1\samples\Java\SharingData\src\com\platform\symphony\samples\SharingData\common\MyOutput.java`
		`%SOAM_HOME%\5.1\samples\Java\SharingData\src\com\platform\symphony\samples\SharingData\common\MyCommonData.java`

	Service	`%SOAM_HOME%\5.1\samples\Java\SharingData\src\com\platform\symphony\samples\SharingData\service\SharingDataService.java`
	Application profile	The service required to compute the input data along with additional application parameters are defined in the application profile: `%SOAM_HOME%\5.1\samples\Java\SharingData\sharingDataJava.xml`
	Output directory	`%SOAM_HOME%\5.1\samples\Java\SharingData`
Linux	Client	`$SOAM_HOME/5.1/samples/Java/SharingData/src/com/platform/symphony/samples/SharingData/client/SharingDataClient.java`

	Input, output, and data objects	`$SOAM_HOME/5.1/samples/Java/SharingData/src/com/platform/symphony/samples/SharingData/common/MyInput.java`
		`$SOAM_HOME/5.1/samples/Java/SharingData/src/com/platform/symphony/samples/SharingData/common/MyOutput.java`
		`$SOAM_HOME/5.1/samples/Java/SharingData/src/com/platform/symphony/samples/SharingData/common/MyCommonData.java`

	Service	`$SOAM_HOME/5.1/samples/Java/SharingData/src/com/platform/symphony/samples/SharingData/service/SharingDataService.java`
	Application profile	The service required to compute the input data along with additional application parameters are defined in the application profile: `$SOAM_HOME/5.1/samples/Java/SharingData/sharingDataJava.xml`
	Output directory	`$SOAM_HOME/5.1/samples/Java/SharingData`

Prerequisites

- Ensure you have installed and started Symphony Developer Edition.
- You should also have completed the following tutorials:
  - Your First Synchronous Symphony Java Client
  - Your First Symphony Java Service

What the sample does

The client creates a session with common data, sends 10 input messages with "Hello Grid!!" through Symphony to the service and retrieves the output synchronously.

The service accesses the common data in onSessionEnter() and stores it in the service container to be accessed during each task invocation. The service then takes input data sent by the client and returns the input data along with "Hello Client !!".

When to use common data

Use common data when the same data is shared among all tasks in a session. You only need to store the data once, and all tasks in a session can access it.

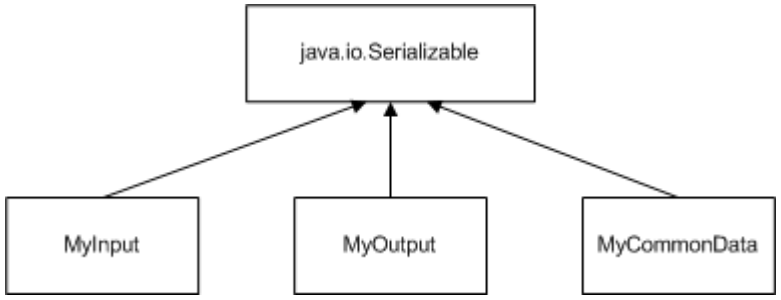
Common data is useful for passing data from a client to a service. The service loads the data when the session is created.

Symphony attempts to use the same service instance for all tasks in a session. A service instance is made available to other sessions only when session workload completes, a session is closed or aborted, or when another session of higher priority is assigned the service instance.

Prepare common data in your client

Declare and implement the Message object

In this tutorial, different classes represent input and output. In addition, we are using an additional class to represent common data.



Use the common data object when creating a session

As in the synchronous client tutorial, initialize the client and connect to the application. Then, create your session to group tasks.

When creating a session, use the common data object to pass data from the client application to the service.

In `SharingDataClient.java`, we create a session and pass in the session attributes including the common data object.

```
...
// Set up our common data to be shared by all task invocations within this session
                MyCommonData commonData = new MyCommonData("Common Data To Be Shared");
                // Set up session attributes
                   SessionCreationAttributes attributes = new SessionCreationAttributes();
                attributes.setSessionName("mySession");
                attributes.setSessionType("ShortRunningTasks");
                attributes.setSessionFlags(Session.SYNC);
                attributes.setCommonData(commonData);
                                // Create a synchronous session
                Session session = null;
                try
                {
                    session = connection.createSession(attributes);
...
```

Continue with your client as usual

Now you can proceed the same way as in the synchronous client tutorial:

- Send input data to be processed
- Retrieve output
- Close the session
- Close the connection
- Catch exceptions
- Uninitialize

Access common data in your service

Define a service container and get data from the session

As in the basic service tutorial, first define a service container. Then retrieve the common data from the session sent by the client by implementing onSessionEnter() before your invoke call.

onSessionEnter() is called once for the duration of the session. The corresponding pair is onSessionLeave ().

In `SharingDataService.java`, we inherited from the ServiceContainer class, and implemented onSessionEnter() to get common data and store it for later use with the session context.

```
...
public class SharingDataService extends ServiceContainer
{
    SharingDataService()
    {
        super();
    }
    /**
     * The middleware triggers the invocation of this handler to bind the Service
     * Instance to its owning session when common data is provided by the Client.
     *
     * If any common data is available for the associated session, it
     * should be accessed in the developer's implementation of this method.
     * Default implementation of this handler does nothing.
     */
    public void onSessionEnter(SessionContext sessionContext) throws SoamException
    {
        // get the current session ID (if needed)
        m_currentSID = sessionContext.getSessionId();

        // populate our common data object
        m_commonData = (MyCommonData)sessionContext.getCommonData();
    }
...
```

Process the input

In this example, we use the common data in our invoke call by formatting the output string. We then set our output message as usual to send common data back with each of the replies.

```
...
public void onInvoke (TaskContext taskContext) throws SoamException
    {
        // We simply echo the data back to the client
        MyOutput myOutput = new MyOutput();
        // estimate and set our runtime
        Date date = new Date();
        myOutput.setRunTime(date.toString());
        // get the input that was sent from the client
        MyInput myInput = (MyInput)taskContext.getTaskInput();
        // echo the ID
        myOutput.setId(myInput.getId());
        // setup a reply to the client
        StringBuffer sb = new StringBuffer();
                sb.append("Client sent : ");
        sb.append(myInput.getString());
        sb.append("\nSymphony replied : Hello Client !! with common data (\"");
        sb.append(m_commonData.getString());
        sb.append("\") for session(");
        sb.append(m_currentSID);
        sb.append(")");
        myOutput.setString(sb.toString());
        // set our output message
        taskContext.setTaskOutput(myOutput);
    }
```

```
...
```

Perform any data cleanup

After processing the input, use the onSessionLeave() call to free the data for the session. onSessionLeave () is called once for every session that is created. In this example, we do not perform any operations in onSessionLeave().

```
...
public void onSessionLeave() throws SoamException
    {
        // We get a chance to do any cleanup for anything we may have done
        // in the onSessionEnter( ) method
    }
...
```

Run the container and catch exceptions

As with the basic service, run the container in the service main and catch exceptions.

```
...
 public static void main(String args[])
    {
        // Return value of our service program
        int retVal = 0;
        try
        {
            // Create the container and run it
            SharingDataService myContainer = new SharingDataService();
            myContainer.run();
        }
        catch (Exception ex)
        {
            // Report exception
            System.out.println("Exception caught:");
            System.out.println(ex.toString());
            retVal = -1;
        }
        // NOTE: Although our service program will return an overall
        // failure or success code it will always be ignored in the
        // current revision of the middleware.
        // The value being returned here is for consistency.
        System.exit(retVal);
    }
...
```

.NET Tutorials

Tutorial: SampleApp: Your first synchronous Symphony C# client and service

Goal

This tutorial guides you through the process of building, packaging, deploying, and running the hello grid sample client and service. It also walks you through the sample application code.

You learn the minimum amount of code that you need to create a client and a service.

At a glance

1. Build the sample client and service
2. Package the sample service
3. Add the application
4. Run the sample client and service
5. Walk through the code

Where to find the documentation

Additional documentation is included in the *%SOAM_HOME%*`\docs` directory, as follows:

Note:

%SOAM_HOME% is an environment variable that represents the Symphony DE installation directory; for example, `C:\SymphonyDE\DE40`.

- .NET API Reference: *%SOAM_HOME%*`\docs\symphonyde\5.1\dotnet\api_reference`
- Platform Symphony Reference: *%SOAM_HOME%*`\docs\symphonyde\5.1\reference_sym`
- Error Reference: *%SOAM_HOME%*`\docs\symphonyde\5.1\error_reference`
- Platform Symphony DE Knowledge Center: *%SOAM_HOME%*`\docs\symphonyde\5.1\index.html`

Build the sample client and service

1. Navigate to *%SOAM_HOME%*`\5.1\samples\DotNet\CS\SampleApp`.
2. Open the Visual C#.NET solution file that is supported by your version of Visual Studio.
3. Build the .NET solution by pressing **ctrl**+**shift**+**B**.

   Compiled executables and libraries are in the `%SOAM_HOME%\5.1\samples\DotNet\CS\SampleApp\output` directory.

Package the sample service

You must package the files required by your service to create a service package.

Note:

Make sure the dlls are included in your service package.

1. Go to the directory that contains the files for the service package:

   ```
   cd %SOAM_HOME%\5.1\samples\DotNet\CS\SampleApp\output
   ```

2. Locate the `SampleServiceDotNetCS.exe` and `Common.dll` files. Add these files to an archive using a compression program such as gzip. Save the archive as `SampleServiceDotNetCS.zip` in the current directory.

Add the application

When you add an application through the DE PMC, you must use the Add Application wizard. This wizard defines a consumer location to associate with your application, deploys your service package, and registers your application. After completing the steps with the wizard, your application should be ready to use.

1. In the DE PMC, click Symphony Workload > Configure Applications.

   The Applications page displays.

2. Select Global Actions > Add/Remove Applications.

   The Add/Remove Application page displays.

3. Select Add an application, then click Continue.

   The Adding an Application page displays.

4. Select Use existing profile and add application wizard, and browse to your application profile.

5. Select your application profile xml file, then click Continue

   For SampleApp, you can find your profile in the following location:

   - .NET:
     - Windows—`%SOAM_HOME%\5.1\samples\DotNet\CS\SampleApp\SampleAppDotNetCS.xml`

   The Service Package location window displays.

6. Browse to the service package you created in .zip format and select it, then, select Continue.

   The Confirmation window displays.

7. Review your selections, then click Confirm.

   The window displays indicating progress. Your application is ready to use.

8. Click Close.

   The window closes and you are now back in the Platform Management Console. Your new application is displayed as enabled.

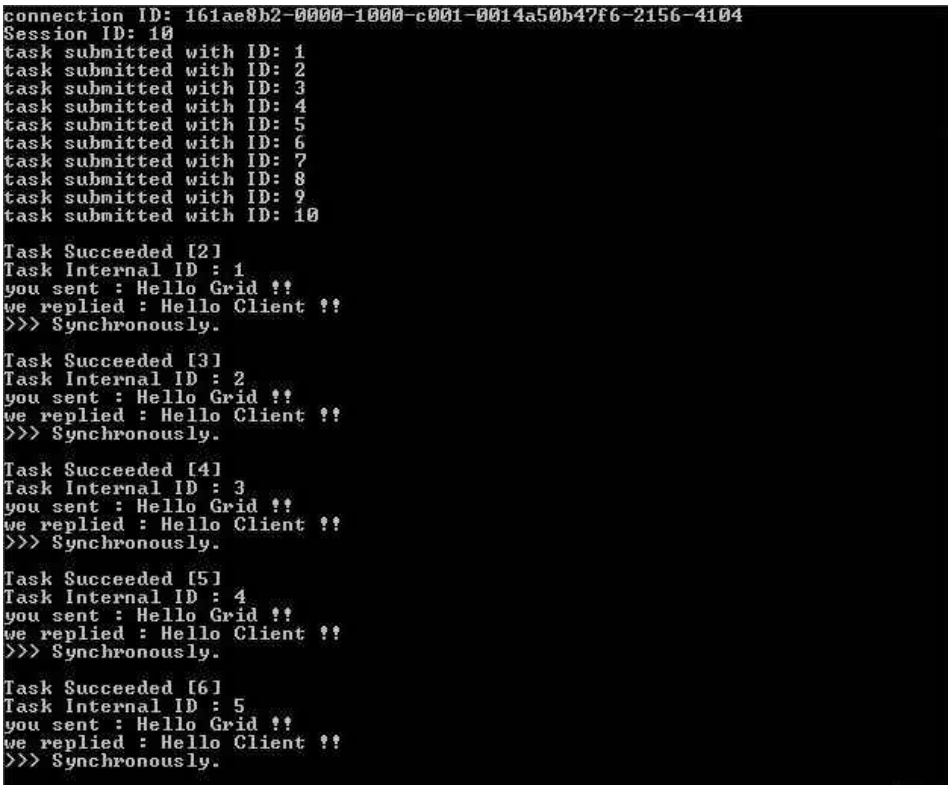
Run the sample client and service

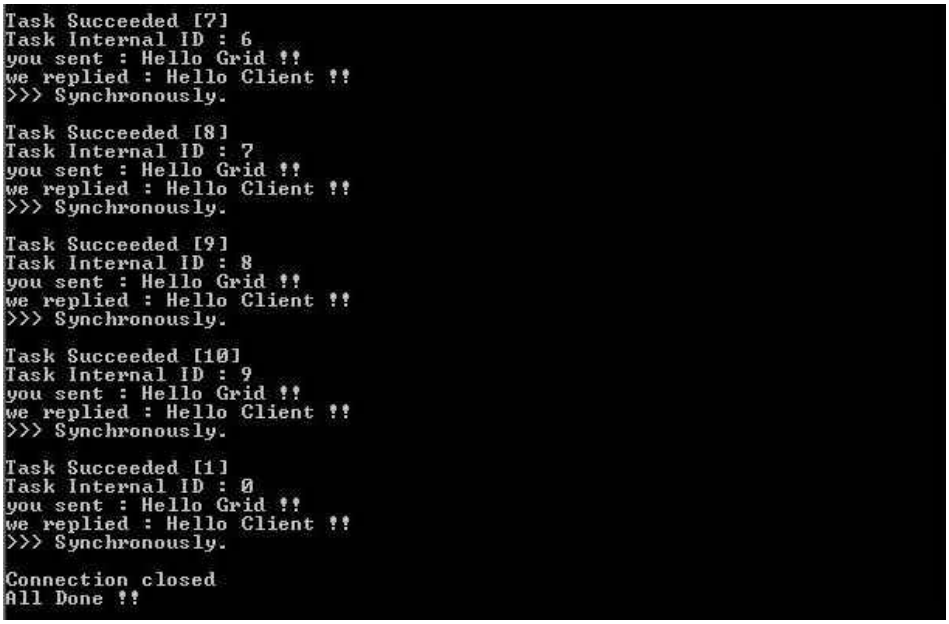
To run the service, run the client application. The service that a client application uses is specified in the application profile.

Before running the sample client, ensure that the SyncClient project is set as the StartUp project in Visual C#.NET.

1. Press F5 to run the application.

The client starts and the system starts the corresponding service. The client displays messages in the console window indicating that it is running.





Walk through the code

You review the sample client application code to learn how you can create a synchronous client application.

Locate the code samples

Solution file (Visual Studio .NET)

`%SOAM_HOME%\5.1\samples\DotNet\CS\SampleApp\sampleApplication.NET.<version>.sln`

or

`%SOAM_HOME%\5.1\samples\DotNet\CS\SampleApp\sampleApplication.NET64.<version>.sln`

where <*version*> is the Visual Studio version.

Client

`%SOAM_HOME%\5.1\samples\DotNet\CS\SampleApp\SyncClient\SyncClient.cs`

Input/output object

`%SOAM_HOME%\5.1\samples\DotNet\CS\SampleApp\Common\MyMessage.cs`

Service

`%SOAM_HOME%\5.1\samples\DotNet\CS\SampleApp\Service\SampleService.cs`

Application profile

The service required to compute the input data along with additional application parameters are defined in the application profile:

`%SOAM_HOME%\5.1\samples\DotNet\CS\SampleApp\SampleAppDotNetCS.xml`

What the sample does

The client application sample sends 10 input messages with the data “Hello Grid !!”, and retrieves the results. The client application is a synchronous client that sends input and blocks the output until all the results are returned.

The service takes input data sent by client applications, returns the input data you have sent and replies "Hello Client !!".

Step 1: Initialize the client

In `SyncClient.cs`, when you initialize, you initialize the Symphony client infrastructure. You initialize once per client.

Important:

Initialization is required. Otherwise, API calls fail.

```
...
SoamFactory.Initialize();
...
```

Step 2: Implement the MyMessage class

Your client application needs to handle data that it sends as input, and output data that it receives from the service.

In `MyMessage.cs`, we implement methods to set and access the data, such as the message string, task ID, and sync flag.

The MyMessage class must be marked with the serializable attribute. The .NET Framework provides the ability to serialize object data for the purpose of passing it by value across application domains. Making the class serializable means that the object can be deconstructed so that it can be passed through the network to the service. Similarly, the object can be reconstructed when it is received by the service.

```
[Serializable]
    public class MyMessage
    {
        public MyMessage()
        {
            m_id     = 0;
            m_isSync = false;
            m_string="";
        }
        public MyMessage(int id, bool isSync, string str)
        {
            m_id     = id;
            m_isSync = isSync;
            m_string = str;
        }
        public bool IsSync
        {
            get
            {
                return m_isSync;
            }
            set
            {
                m_isSync = value;
            }
        }
        public int Id
        {
            get
            {
                return m_id;
            }
            set
            {
                 m_id = value;
            }
        }
        public string StringMessage
        {
            get
            {
                 return m_string;
            }
            set
            {
                 m_string = value;
            }
        }
...
```

Step 3: Connect to an application

To send data to be calculated in the form of input messages, you connect to an application.

You specify an application name, a user name, and password. The application name must match that defined in the application profile.

For Symphony Developer Edition, there is no security checking and login credentials are ignored—you can specify any user name and password. Security checking is done however, when your client application submits workload to the actual grid.

The default security callback encapsulates the callback for the user name and password.

```
...
// Set up application specific information to be supplied to Symphony
                String applicationName="SampleAppDotNetCS";
                // Set up application authentication information using
                // the default security provider
                DefaultSecurityCallback securityCb = new
                    DefaultSecurityCallback("Guest", "Guest");
                Connection connection = null;
                try
                {
                    // Connect to the specified application
                    connection = SoamFactory.Connect(applicationName, securityCb);
                                    // Retrieve and print our connection ID
                    Console.WriteLine("connection ID: " + connection.Id );
...
```

Tip:

When you connect, a connection object is returned. You can retrieve the connection ID from the object. Save the connection ID. Should the client application fail, you can use the connection ID to reconnect to Symphony Developer Edition.

Important:

It should be emphasized that the creation and usage of the connection object be scoped in a try-finally block. The finally block, with the connection.Close() method, ensures that the connection is always closed whether exceptional behavior occurs or not. Failure to close the connection causes the connection to continue to occupy middleware resources.

```
...
try
{
         // Connect to the specified application
                    connection = SoamFactory.Connect(applicationName, securityCb);
       ...
}
                finally
                {
                    // Mandatory connection close
                    if (connection != null)
                    {
                        connection.Close();
                        Console.WriteLine("Connection closed");
                    }
                 }
...
```

Step 4: Create a session to group tasks

A session is a way of logically grouping tasks that are sent to a service for execution. The tasks are sent and received synchronously.

When creating a synchronous session, you need to specify the session attributes by using the SessionCreationAttributes object. In this sample, we create a SessionCreationAttributes object called attributes and set three parameters in the object.

```
                    ...
try
{
    // Set up session attributes
    SessionCreationAttributes attributes = new SessionCreationAttributes();
    attributes.SessionName ="mySession";
    attributes.SessionType ="ShortRunningTasks";
    attributes.SessionFlags = SessionFlags.AliasSync;
    // Create a synchronous session
    session = connection.CreateSession(attributes);
                    ...
```

In this example, note that:

- The first parameter is the session description. This is optional. The session description can be any descriptive name you want to assign to your session. It is for information purposes, such as in the command-line interface.
- The second parameter is the session type. The session type is optional. You can leave this parameter blank and system default values are used for your session.

  Important:

  The session type must be the same session type as defined in your application profile.

  In the application profile, with the session type, you define characteristics for the session.
- The third parameter is the session flag. When creating a synchronous session, set the flag to SessionFlags.AliasSync. This flag indicates to Symphony that this is a synchronous session.

Important:

As is the case with the connection object, the creation and usage of the session object, i.e., sending and receiving data, must be scoped in a try-finally block. The finally block, with the session.Close() method, ensures that the session is always closed whether exceptional behavior occurs or not. Failure to close the session causes the session to continue to occupy middleware resources.

```
    ...
    try
    {
          session = connection.CreateSession(attributes);
    ...
    }
    finally
    {
          // Mandatory session close
          if (session != null)
          {
               session.Close();
               Console.WriteLine("Session closed");
          }
    }
    ...
```

Step 5: Send input data to be processed

In this step, we create 10 input messages to be processed by the service. We call the MyMessage constructor and pass three input parameters: ID (taskCount), the Boolean value (true) to indicate synchronous

communication, and a message string ("Hello Grid !!"). When a message is sent, a task input handle is returned. This task input handle contains the ID for the task that was created for this input message.

```
...
for (int taskCount = 0; taskCount < numTasksToSend; taskCount++)
{
          // Create a message
          MyMessage inputMessage = new MyMessage(taskCount, true, "Hello Grid !!");
          // Set task submission attributes
          TaskSubmissionAttributes taskAttr = new TaskSubmissionAttributes();
          taskAttr.SetTaskInput(inputMessage);
          // Send it
          TaskInputHandle input = session.SendTaskInput(taskAttr);
          // Retrieve and print task ID
          Console.WriteLine("task submitted with ID: " + input.Id);
}
...
```

Step 6: Retrieve the output

Pass the number of tasks to the FetchTaskOutput() method to retrieve the output messages that were produced by the service instance. This method blocks until the output for all tasks is retrieved. The return value is an enumeration that contains the completed task results.

Iterate through the task results and extract the messages using the GetTaskOutput() method. Display the task ID, internal ID (taskCount), and the output message.

```
...
      EnumItems enumItems = session.FetchTaskOutput( (ulong) numTasksToSend);
      // inspect results
      foreach(TaskOutputHandle output in enumItems)
      {
          // check for success of task
          if ( output.IsSuccessful == true )
          {
              // get the message returned from the service
              MyMessage outputMessage = output.GetTaskOutput() as MyMessage;
              if(outputMessage == null)
              {
                 throw new SoamException("Received unexpected object type for task output.");
              }
              // display content of reply
              Console.WriteLine("Task Succeeded [" + output.Id + "]" );
              Console.WriteLine("Task Internal ID : " + outputMessage.Id );
              Console.WriteLine(outputMessage.StringMessage );
          }
          else
          {
              // get the exception associated with this task
              SoamException ex = output.Exception;
              Console.WriteLine( ex.ToString() );
          }
      }
 ...
```

Step 7: Catch exceptions

Any exceptions thrown take the form of SoamException. Catch all Symphony exceptions to know about exceptions that occurred in the client application, service, and middleware.

The sample code above catches exceptions of type SoamException.

Step 8: Uninitialize

Always uninitialize the client API at the end of all API calls. If you do not call uninitialize, the client API remains in an undefined state and resources used by the client are held indefinitely.

Important:

Once you uninitialize, all objects become invalid. For example, you can no longer create a session or send an input message.

```
...
SoamFactory.Uninitialize();
...
```

Step 9: Define a service container

For a service to be managed by Symphony, it needs to be in a container object. This is the service container.

In `SampleService.cs`, SampleServiceContainer inherits from the base class ServiceContainer.

```
...
class SampleServiceContainer : ServiceContainer
{
...
```

Step 10: Process the input

Symphony calls OnInvoke() on the service container once per task. Once you inherit from the ServiceContainer class, implement handlers so that the service can function properly. This is where the calculation is performed.

To gain access to the data set from the client, you call the GetTaskInput() method on the task context. The middleware is responsible for placing the input into the taskContext object.

The task context contains all information and functionality that is available to the service during an OnInvoke() call in relation to the task that is being processed.

In this sample, we use the StringBuilder object to build the output message, which includes the input message that is echoed back to the client. Since we are using the same service for sync and async clients, the if statement is used to indicate that the message was sent from a sync client. When the string in the output message is completely assembled, pass it to the SetTaskOutput() method, which sets the task output message that is sent to the client.

```
...
public override void OnInvoke(TaskContext taskContext)
{
       // get the input that was sent from the client
       MyMessage inputMsg = taskContext.GetTaskInput() as MyMessage;
       if(inputMsg == null)
       {
           throw new SoamException("Have got wrong type of outputMessage object.");
       }
       // We simply echo the data back to the client
       MyMessage outputMsg = new MyMessage();
       outputMsg.Id = inputMsg.Id;
       StringBuilder reply = new StringBuilder();
       reply.Append("you sent : ");
       reply.Append(inputMsg.StringMessage);
       reply.Append("\nwe replied : Hello Client !!\n>>> ");
       if (inputMsg.IsSync)
       {
           reply.Append("Synchronously.\n");
       }
       else
       {
           reply.Append("Asynchronously.\n");
       }
       outputMsg.StringMessage = reply.ToString();

       // set our output message
       taskContext.SetTaskOutput(outputMsg);
}
...
```

Step 11: Run the container

The service is implemented within an executable. As a minimum, we need to create within our main function an instance of the service container and run it.

Note that our service is started by parameters.

```
...
static int Main(string[] args)
{
       // Return value of our service program
       int returnValue = 0;
       try
       {
              // Create a new service container and run it
              SampleServiceContainer myContainer = new SampleServiceContainer();
              myContainer.Run();
       }
...
```

Step 12: Catch exceptions

Catch exceptions in case the container fails to start running.

```
...
       catch( Exception ex )
       {
              // report exception
              Console.WriteLine( "Exception caught ... " + ex.ToString() );
              returnValue = -1;
       }
...
```

Tutorial: SampleApp: Developing an asynchronous Symphony C# client

Goal

The purpose of an asynchronous client is to get the output as soon as it is available. The client thread does not need to be blocked once the input data is sent and can perform other actions.

In this tutorial, you learn how to convert a synchronous client into asynchronous.

At a glance

1. Build the sample client and service
2. Package the sample service and add the application
3. Run the sample client and service
4. Walk through the code

Where to find the documentation

Additional documentation is included in the `%SOAM_HOME%\docs` directory, as follows:

Note:

%SOAM_HOME% is an environment variable that represents the Symphony DE installation directory; for example, `C:\SymphonyDE\DE51`.

- .NET API Reference: `%SOAM_HOME%\docs\symphonyde\5.1\dotnet\api_reference`
- Platform Symphony Reference: `%SOAM_HOME%\docs\symphonyde\5.1\reference_sym`
- Error Reference: `%SOAM_HOME%\docs\symphonyde\5.1\error_reference`
- Platform Symphony DE Knowledge Center: `%SOAM_HOME%\docs\symphonyde\5.1\index.html`

Build the sample client and service

1. Navigate to `%SOAM_HOME%\5.1\samples\DotNet\CS\SampleApp`.
2. Open the Visual C#.NET solution file that is supported by your version of Visual Studio.
3. Build the solution files by pressing **ctrl**+**shift**+**B**.

   Compiled executables and libraries are in the `%SOAM_HOME%\5.1\samples\DotNet\CS\SampleApp\output` directory.

Package the sample service and add the application

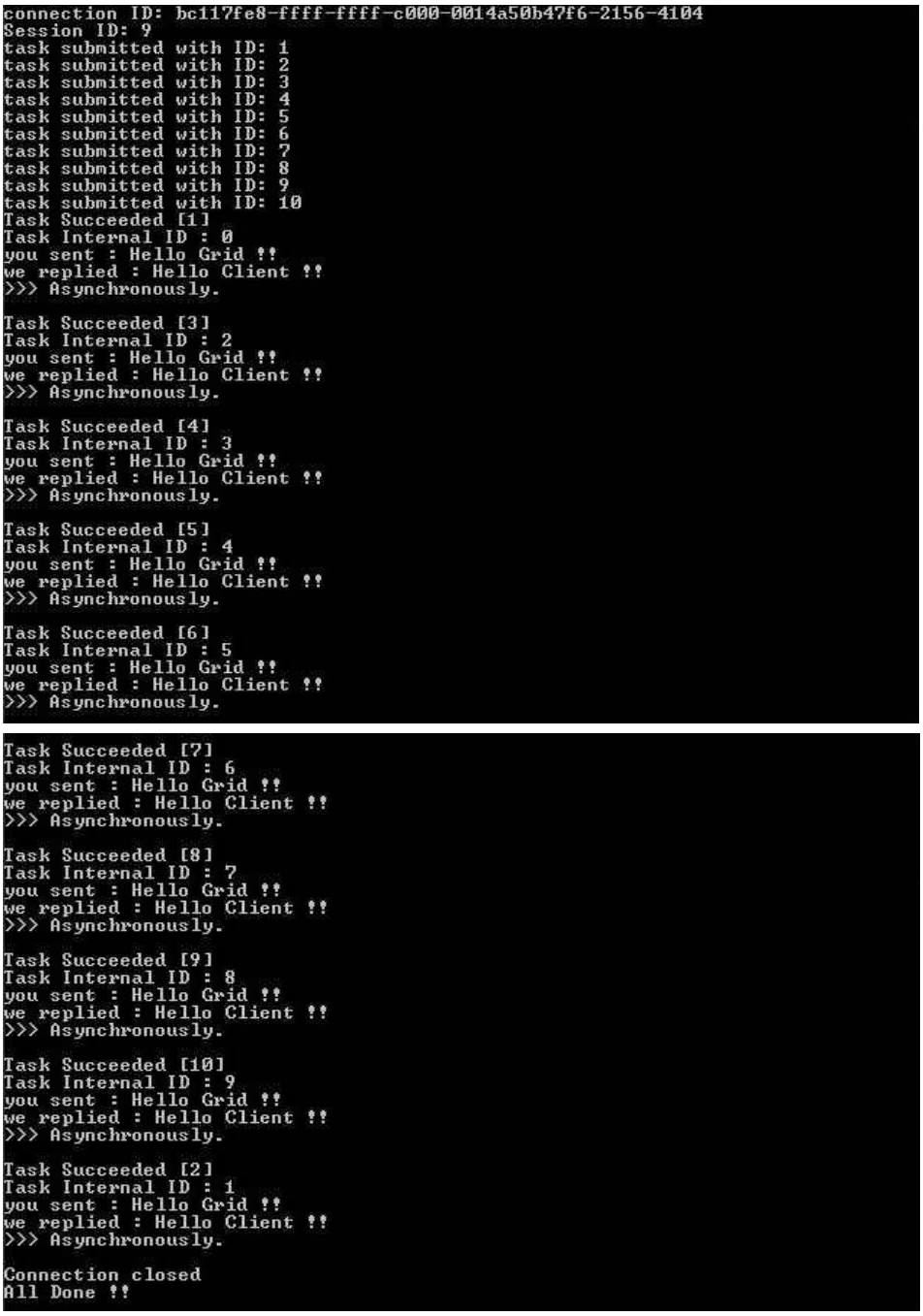
If you have not already packaged and added the sample application, refer to the synchronous client tutorial for more details.

Run the sample client and service

To run the service, run the client application. The service that a client application uses is specified in the application profile. Before running the sample client, ensure that the AsyncClient project is set as the StartUp project in Visual C#.NET.

1. Press F5 to run the application.

   The client starts and the system starts the corresponding service. The client displays messages in the console window indicating that it is running.



Walk through the code

Review the sample client application code to learn how you can understand the differences between a synchronous client and an asynchronous client.

Locate the code samples

Solution file (Visual Studio)

```
%SOAM_HOME%\5.1\samples\DotNet\CS\SampleApp
\sampleApplication.NET.<version>.sln
```

or

```
%SOAM_HOME%\5.1\samples\DotNet\CS\SampleApp
\sampleApplication.NET64.<version>.sln
```

where <*version*> is the version of Visual Studio.

Client

```
%SOAM_HOME%\5.1\samples\DotNet\CS\SampleApp\AsyncClient\AsyncClient.cs
```

Input/output object

```
%SOAM_HOME%\5.1\samples\DotNet\CS\SampleApp\Common\MyMessage.cs
```

Service

```
%SOAM_HOME%\5.1\samples\DotNet\CS\SampleApp\Service\SampleService.cs
```

Application profile

The service required to compute the input data along with additional application parameters are defined in the application profile:

```
%SOAM_HOME%\5.1\samples\DotNet\CS\SampleApp\SampleAppDotNetCS.xml
```

What the sample does

The client application sample sends 10 input messages with the data "Hello Grid !!" and retrieves the results.

Results are returned asynchronously with a callback interface provided by the client to the API. Methods on this interface are called from threads within the API when certain events occur. In the sample, the events are:

- When there is an error at the session level
- When results return from the middleware

Code differences between synchronous and asynchronous clients

An asynchronous client is very similar to a synchronous client. The only differences are:

- You need to specify a callback when creating a session
- You specify a different flag to indicate asynchronous when you create a session
- Retrieval of replies

Step 1: Initialize the client and implement the MyMessage class

As in the synchronous client tutorial, initialize the client and implement the MyMessage class to handle the input/output data; refer to Your First Synchronous Symphony C# Client and Service, specifically:

- Step 1: Initialize the client on page 93

- Step 2: Implement the MyMessage class on page 93

Step 2: Implement the response handler method to retrieve output messages

With an asynchronous client, when a task is completed by the service, there must be a means of communicating this status back to the client. The response handler is implemented for this purpose. It is called by the middleware each time a service completes a task.

In this sample, the AsyncClientOnResponse() method is the response handler. The method accepts the TaskOutputHandle as an input argument, which is passed to the method by the middleware whenever the respective task has completed.

Extract the message from the task result using the GetTaskOutput() method. Display the task ID, internal ID (taskCount), and output message.

Increment the m_numReceivedTasks variable. Use the lock keyword to ensure that another thread does not try to increment the variable while it is being accessed.

The m_eventOccured.Set() method releases the waiting main execution thread of the client.

```
private void AsyncClientOnResponse( TaskOutputHandle output )
{
       // check for success of task
       if ( output.IsSuccessful == true )
       {
           // get the message returned from the service
           MyMessage outputMessage = output.GetTaskOutput() as MyMessage;
           if(outputMessage == null)
                {
                    throw new SoamException("Have got wrong type of outputMessage object.");
                }
           // display content of reply
           Console.WriteLine("Task Succeeded [" + output.Id + "]" );
           Console.WriteLine("Task Internal ID : " + outputMessage.Id );
           Console.WriteLine(outputMessage.StringMessage );
       }
       else
       {
           // get the exception associated with this task
           SoamException ex = output.Exception;
           Console.WriteLine( ex.ToString() );
       }

       lock(this)
       {
           m_numReceivedTasks++;
       }
       m_eventOccured.Set();
}
```

Step 3: Implement the exception handler method (callback)

The exception handler method is called by the API when an exception of type SoamException occurs within the scope of a session.

Print out the exception message and set the Boolean error flag (m_noErrorReported) to false. Use the lock keyword to ensure that another thread does not try to set the flag while it is being accessed.

```
private void AsyncClientOnException(SoamException exception)
{
       Console.WriteLine( "Soam exception caught ... " + exception.ToString());
       lock(this)
       {
            m_noErrorReported = false;
       }
       m_eventOccured.Set();
}
```

Step 4: Connect to an application

To send data to be calculated in the form of input messages, you connect to an application; refer to Step 3: Connect to an application of the synchronous client tutorial.

Step 5: Create a session to group tasks

A session is a way of logically grouping tasks that are sent to a service for execution. The tasks are sent and received asynchronously.

When creating a session, you need to specify the session attributes by using the SessionCreationAttributes object. In this sample, we create a SessionCreationAttributes object called attributes and set four parameters in the object.

In this example, we set the following parameters:

- The first parameter is the session description. This is optional. The session description can be any descriptive name you want to assign to your session. It is for information purposes, such as in the command-line interface.
- The second parameter is the session type. The session type is optional. You can leave this parameter blank and system default values are used for your session.

Important:

The session type must be the same session type as defined in your application profile.

In the application profile, with the session type, you define characteristics for the session.

- The third parameter is the session flag. When creating an asynchronous session, set the flag to SessionFlags.ReceiveAsync. This flag indicates to Symphony that this is an asynchronous session.
- The fourth parameter is the callback object.

```
...
try
{
// Set up session attributes
                        SessionCreationAttributes attributes = new
                        SessionCreationAttributes();
                        attributes.SessionName="mySession";
                        attributes.SessionType="ShortRunningTasks";
                        attributes.SessionFlags = SessionFlags.ReceiveAsync;
                        attributes.SessionCallback = callback;
                        // Create an asynchronous session
                        session = connection.CreateSession(attributes);
...
}
...
```

Step 6: Associate event handlers with the events

Associate the event handler method (AsyncClientOnResponse) with the OnResponse event; refer to Step 2: Implement the response handler method to retrieve output messages. This is necessary so that the OnResponse event knows which method to execute when the event is triggered. The method is called by the API whenever a task response is ready. Similarly, associate the AsyncClientOnException() method with the OnException event to handle exceptions of type SoamException if they occur. This method is called by the API when an exception occurs within the scope of the session; refer to Step 3: Implement the exception handler method (callback).

```
...
 // Create a SessionCallback instance and register event handlers
 SessionCallback callback = new SessionCallback();
 callback.OnResponse += new SessionCallback.ResponseHandler(AsyncClientOnResponse);
 callback.OnException += new SessionCallback.ExceptionHandler(AsyncClientOnException);
...
```

Step 7: Send input data to be processed

In this step, we create 10 input messages to be processed by the service. We call the MyMessage constructor and pass three input parameters: ID (taskCount), the Boolean value (false) to indicate asynchronous communication, and a message string ("Hello Grid !!"). When a message is sent, a task input handle is returned. This task input handle contains the ID for the task that was created for this input message.

```
...
int numTasksToSend = 10;
for (int taskCount = 0; taskCount < numTasksToSend; taskCount++)
{
       // Create a message
       MyMessage inputMessage = new MyMessage(taskCount, false, "Hello Grid !!");
       TaskSubmissionAttributes taskAttr = new TaskSubmissionAttributes();
       taskAttr.SetTaskInput(inputMessage);
       // send it
       TaskInputHandle input = session.SendTaskInput(inputMessage);
       // retrieve and print task ID
       Console.WriteLine( "task submitted with ID: " + input.Id);
}
...
```

Step 8: Wait for replies before closing the session

After all 10 tasks (messages) have been sent to the service, the main client execution thread must wait for all tasks to be processed before closing the session. As each task is completed by the service, the m_numReceivedTasks variable is incremented; refer to Step 2: Implement the response handler method to retrieve output messages. The WaitForComplete() method is used to suspend the main client execution thread until all messages are received. The method contains a loop that checks if the number of replies equals the total number of tasks sent; if they are not equal, the thread blocks by calling m_eventOccured.WaitOne() until it is signalled to resume execution. The thread is released by calling m_eventOccured.Set() each time a task is completed or if an exception occurs. When all the replies have been received, close the session.

Important:

As is the case with the connection object, the creation and usage of the session object, i.e., sending and receiving data, must be scoped in a try-finally block. The finally block, with the session.Close() method, ensures that the session is always closed whether exceptional behavior occurs or

not. Failure to close the session causes the session to continue to occupy middleware resources.

```
...
private void WaitForComplete(int taskCount)
{
       while(true)
       {
           bool shouldWait;
           lock(this)
           {
               shouldWait = (m_numReceivedTasks < taskCount) && m_noErrorReported;
           }
           if (shouldWait)
           {
               m_eventOccured.WaitOne();
           }
           else
           {
               break;
           }
       }
}
...
```

```
...
}finally
{
     // mandatory session close
     if (session != null)
     {
         session.Close();
     }
...
```

Step 9: Uninitialize

Always uninitialize the client API at the end of all API calls. If you do not call uninitialize, the client API remains in an undefined state and resources used by the client are held indefinitely.

Important:

Once you uninitialize, all objects become invalid. For example, you can no longer create a session or send an input message.

```
...
SoamFactory.Uninitialize();
...
```

Tutorial: SharingData: Developing a C# client and service to share data among tasks

Goal

This tutorial walks you through how to develop a client application and service to share data among all tasks in a session. The data is shared by all invocations of tasks within the same session.

You lean how to use different data objects for input, output, and common data.

At a glance

Before you begin, ensure you have installed and started Platform Symphony Developer Edition.

1. Build the sample client and service
2. Package the sample service
3. Add the application
4. Run the sample client and service
5. Walk through the code

Build the sample client and service

1. Navigate to *%SOAM_HOME%*`\5.1\samples\DotNet\CS\SampleApp`.
2. Open the Visual C#.NET solution file that is supported by your version of Visual Studio.
3. Build the .NET solution by pressing **ctrl**+**shift**+**B**.

   Compiled executables and libraries are in the `%SOAM_HOME%\5.1\samples\DotNet\CS\SharingData\output` directory.

Note:

%SOAM_HOME% is an environment variable that represents the Symphony DE installation directory; for example, `C:\SymphonyDE\DE51`.

Package the sample service

You must package the files required by your service to create a service package.

Note:

Make sure the dlls are included in your service package.

1. Go to the directory that contains the files for the service package:

   ```
   cd %SOAM_HOME%\5.1\samples\DotNet\CS\SharingData\output
   ```

2. Locate the `SharingDataServiceDotNetCS.exe` and `Common.dll` files. Add these files to an archive using a compression program such as gzip. Save the archive as `SharingDataServiceDotNetCS.zip` in the current directory.

Add the application

When you add an application through the DE PMC, you must use the Add Application wizard. This wizard defines a consumer location to associate with your application, deploys your service package, and registers your application. After completing the steps with the wizard, your application should be ready to use.

1. Click Symphony Workload > Configure Applications.

   The Applications page displays.

2. Select Global Actions > Add/Remove Applications.

   The Add/Remove Application page displays.

3. Select Add an application, then click Continue.

The Adding an Application page displays.

4. Select Use existing profile and add application wizard, and browse to your application profile.
5. Select your application profile xml file, then click Continue

   For SharingData, you can find your profile in the following location:

   - .NET:
     - `%SOAM_HOME%\5.1\samples\DotNet\CS\SharingData\SharingDataDotNetCS.xml`

   The Service Package location window displays.
6. Browse to the service package you created in .gz, .zip, or tar.gz format and select it, then, select Continue.

   The Confirmation window displays.
7. Review your selections, then click Confirm.

   The window displays indicating progress. Your application is ready to use.
8. Click Close.

   The window closes and you are now back in the Platform Management Console. Your new application is displayed as enabled.

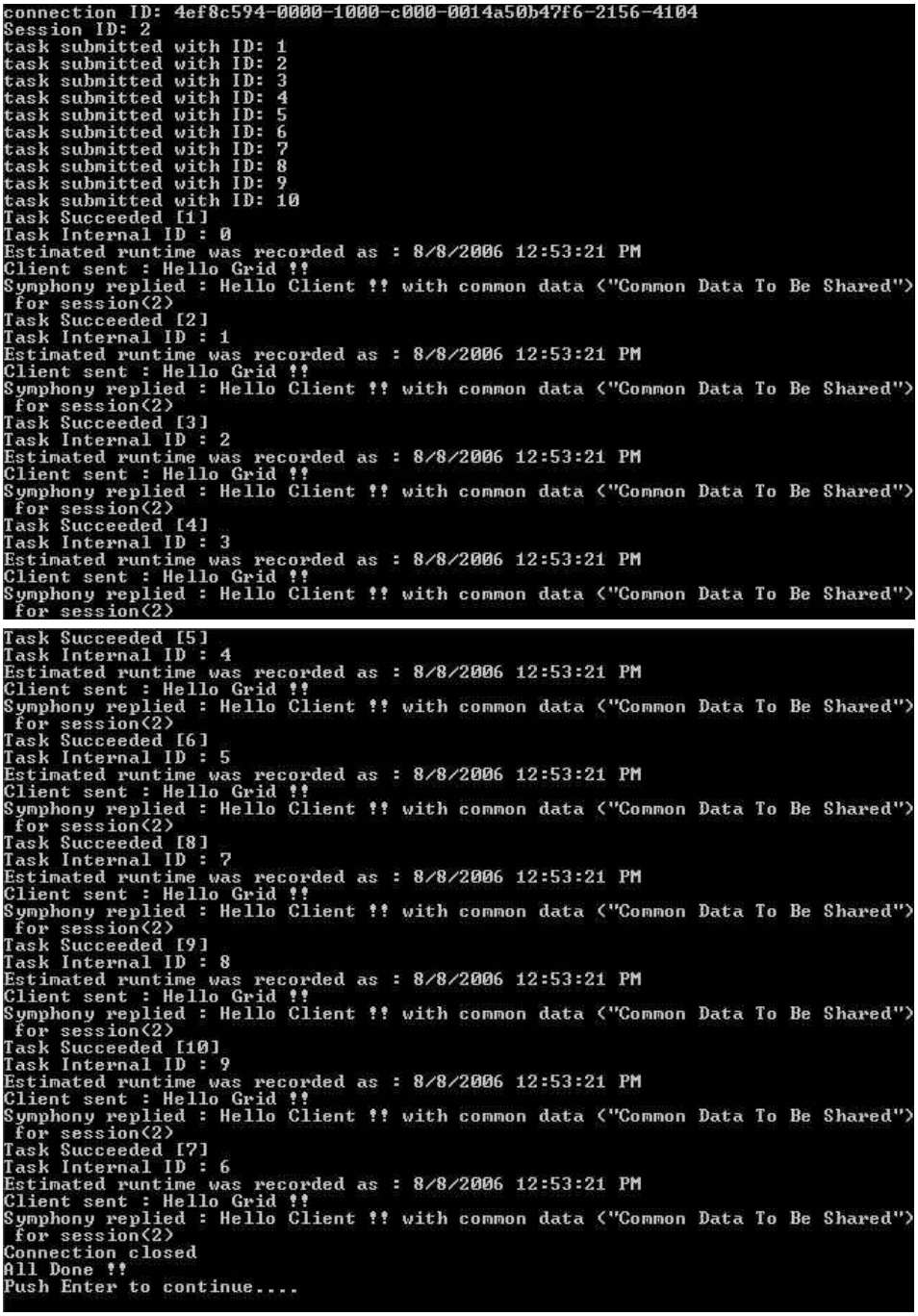
Run the sample client and service

To run the service, run the client application. The service that a client application uses is specified in the application profile.

Before running the sample client, ensure that the Client project is set as the StartUp project in Visual C#.NET.

1. Press F5 to run the application.

   The client starts and the system starts the corresponding service. The client displays messages in the console window indicating that it is running.



Walk through the code

Review the sample code to learn how you can create a client and service that uses common data.

Locate the code samples

Solution file (Visual Studio)

```
%SOAM_HOME%\5.1\samples\DotNet\CS\SharingData
\sharing_data_<version>.sln
```

or

```
sharing_data64_<version>.sln
```

where <*version*> is the version of Visual Studio.

Client

```
%SOAM_HOME%\5.1\samples\DotNet\CS\SharingData\Client\SyncClient.cs
```

Input, output, and data objects

```
%SOAM_HOME%\5.1\samples\DotNet\CS\SharingData\Common\
```

Service

```
%SOAM_HOME%\5.1\samples\DotNet\CS\SharingData\Service\SampleService.cs
```

Application profile

The service required to compute the input data along with additional application parameters are defined in the application profile:

```
%SOAM_HOME%\5.1\samples\DotNet\CS\SharingData\SharingDataDotNetCS.xml
```

What the samples do

The client creates a session with common data. It sends 10 input messages, and retrieves the output. The client then outputs "Hello Grid !!".

The service takes input data sent by client applications, returns the input data you have sent and replies "Hello Client !!" followed by the common data appended to the message. The service uses OnSessionEnter () to define attributes global to the session.

When to use common data

Common data is data that can be made available to service instances for the duration of a session.

Use common data when you need to set up the initial state of a service, and you only want to do it once, not on every task. Common data is useful for passing data from a client to a service. The service loads the data when the session is created.

You can use common data, for example, to set the environment in the service that is common to all tasks in a session. This way you only need to set the environment once, when the session is created.

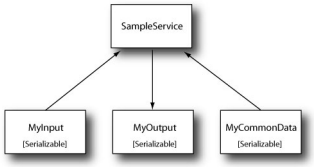
Symphony attempts to use the same service instance for all tasks in a session. A service instance is made available to other sessions only when session workload completes, a session is closed or aborted, or when another session of higher priority is assigned the service instance.

Prepare common data in your client

Declare and implement the Message objects

In the synchronous client tutorial, input and output message objects were the same object. In this tutorial, different objects represent input and output. In addition, we are creating an additional object to represent common data.

Remember to mark your message and data handler classes with the Serializable attribute. This allows the objects to be serialized for transfer across the network.



Use the common data object when creating a session

As in the synchronous client tutorial, initialize the client and connect to the application. Then, create your session to group tasks.

When creating a session, use the common data object to pass data from the client application to the service.

In `SyncClient.cs`, we create a session and pass the common data object via the SessionCreationAttributes object.

```
...
                    // Set up our common data to be shared by all task
                    // invocations within this session
                    MyCommonData commonData = new MyCommonData();
                    commonData.StringMessage="Common Data To Be Shared";
                    Session session = null;
                    try
                    {
                        // Set up session attributes
                        SessionCreationAttributes attributes = new
                        SessionCreationAttributes();
                        attributes.SessionName="mySession";
                        attributes.SessionType="ShortRunningTasks";
                        attributes.SessionFlags = SessionFlags.AliasSync;
                        attributes.SetCommonData(commonData);
                        // Create a synchronous Session
                        session = connection.CreateSession(attributes);
...
```

Continue with your client as usual

Now you can proceed the same way as in the synchronous client tutorial:

- Send input data to be processed
- Retrieve output
- Close the session
- Close the connection
- Catch exceptions
- Uninitialize

Access common data in your service

Define a service container and get data from the session

As in the basic service tutorial, first define a service container. Then retrieve the common data sent by the client by implementing OnSessionEnter(). This method is called once by the middleware for the duration of the session to bind the service instance to the session.

In `SampleService.cs`, we use OnSessionEnter() to get common data and store it for later use within the scope of the session.

```
...
public override void OnSessionEnter(SessionContext sessionContext)
{
       // get the current session ID (if needed)
       m_currentSID = sessionContext.SessionId;

       // populate our common data object
       m_commonData = sessionContext.GetCommonData() as MyCommonData;
       if(m_commonData == null)
       {
           throw new SoamException("Have got wrong type of CommonData object.");
       }
}
...
```

Process the input

In this example, each time the OnInvoke() method is called by the middleware, we append the common data to the output string. We then set our output message as usual to send common data back with each of the replies.

```
...
public override void OnInvoke(TaskContext taskContext)
{
            // estimate and set our runtime
            MyOutput outMsg = new MyOutput();
            outMsg.RunTime = DateTime.Now.ToString();
            // Get the input that was sent from the client
            MyInput inMsg = taskContext.GetTaskInput() as MyInput;

            if(inMsg == null)
            {
                throw new SoamException("The service attempted to access the wrong type
                of InputMessage object.");
            }
            // We simply echo the data back to the client
            outMsg.Id = inMsg.Id;
            StringBuilder reply = new StringBuilder();
            reply.Append("Client sent : ");
            reply.Append(inMsg.StringMessage);
            reply.Append("\nSymphony replied : Hello Client !! with common data (\"");
            reply.Append(m_commonData.StringMessage);
            reply.Append("\") for session(");
            reply.Append(String.Format("{0}", m_currentSID));
            reply.Append(")");
            outMsg.StringMessage = reply.ToString();
            // Set our output message
            taskContext.SetTaskOutput(outMsg);
}
```

```
...
```

Perform any data cleanup

After processing the input, use the OnSessionLeave() call to free the data for the session. The OnSessionLeave() method is called once by the middleware for every session that is created.

```
...
public override void OnSessionLeave()
{
       // free our data
       m_currentSID = null;
       m_commonData = null;
}
...
```

Run the container and catch exceptions

As with the basic service, run the container in the service main and catch exceptions.

```
...
static int Main(string[] args)
{
       // Return value of our service program
       int returnValue = 0;
       try
       {
            // Create a new service container and run it
            SampleServiceContainer myContainer = new SampleServiceContainer();
            myContainer.Run();
       }
       catch( Exception ex )
       {
            // report exception
            Console.WriteLine( "Exception caught ... " + ex.ToString() );
            returnValue = -1;
       }
       return returnValue;
}
...
```

Cross-language Tutorials

Developing cross-language clients and services

Tutorial: CrossLanguage: Developing cross-language clients and services

Goal

This tutorial walks you through using Symphony serialization when developing client applications and services. Symphony serialization allows clients and services written in different programming languages to communicate with each other. For example, you can use a C++ client with a Java service.

You should also use Symphony serialization if you are concerned with performance and memory usage.

In this tutorial, you build samples in C++, Java, and .NET, package and deploy the service in either language, and use the C++, Java, COM, and .NET clients to submit work to the service.

At a glance

1. Build the samples
2. Package the service
3. Add the application
4. Run the sample clients
5. Walk through the code

Prerequisites

- Ensure that you have installed and started Symphony DE.
- You should also have completed the following tutorials in either C++, Java, or .NET:
  - Your First Synchronous Symphony Client
  - Your First Symphony Service

Build the samples

The following section provides instructions for building the samples in C++, Java, COM, and .NET.

You need to build all samples to have cross-language clients and services.

Build the C++ sample client and service

On Windows

You can build client application and service samples at the same time.

1. In `%SOAM_HOME%\5.1\samples\CrossLanguage\CPP`, locate workspace file `CrossLanguageSampleCPP_vc6.dsw`, or one of the solution files supported by your version of Visual Studio.

2. Load the file into Visual Studio and build it.

On Linux

You can build client application and service samples at the same time.

1. Change to the `$SOAM_HOME/conf` directory.
2. Set the environment:
   - For `csh`, enter
     ```
     source cshrc.soam
     ```
   - For `bash`, enter
     ```
     . profile.soam
     ```
3. Compile using the Makefile located in `$SOAM_HOME/5.1/samples/CrossLanguage/CPP`:
   ```
   make
   ```

Build the Java sample client and service

On Windows

Compile with the .bat file

You can build client application and service samples at the same time.

1. Change to the `%SOAM_HOME%\5.1\samples\CrossLanguage\Java\` directory and run the .bat file:
   ```
   build.bat
   ```

Compile with the Ant build file

You can build client application and service samples at the same time.

1. Change to the `%SOAM_HOME%\5.1\samples\CrossLanguage\Java\` directory and run the build command:
   ```
   ant
   ```

Compile in Eclipse

To compile in Eclipse, see "Symphony plug-in for Eclipse" in the *Application Development Guide*.

On Linux

Compile with the Makefile

You can build client application and service samples at the same time.

1. Change to the `$SOAM_HOME/conf` directory.
2. Set the environment:
   - For csh, enter
     ```
     source cshrc.soam
     ```
   - For bash, enter
     ```
     . profile.soam
     ```

3. Change to the `$SOAM_HOME/5.1/samples/CrossLanguage/Java` directory and run the command:

```
make
```

Compile with the Ant build file

You can build client application and service samples at the same time.

1. Change to the `$SOAM_HOME/conf` directory.
2. Set the environment:
   - For csh, enter

```
source cshrc.soam
```

   - For bash, enter

```
. profile.soam
```

3. Change to the $SOAM_HOME/5.1/samples/CrossLanguage/Java directory and run the command:

```
ant
```

Compile in Eclipse

To compile in Eclipse, see "Symphony plug-in for Eclipse" in the *Application Development Guide.*

Build the .NET sample client and service

1. Double-click the appropriate Visual C#.NET solution file located in the `$SOAM_HOME/5.1/samples/CrossLanguage/DotNet`.
2. Build the solution.

Build the COM client

1. Double-click to open VB_Clients.vbp located in the `%SOAM_HOME%\4.0\samples\CrossLanguage\COM\Client` directory.
2. In the Microsoft Visual Basic, open the `SyncClient.frm` code and uncomment `api.Uninitialize`, which is under `Form_Unload`. Close the form.
3. In the Microsoft Visual Basic, build the COM client by clicking File>Make ComApiClients.exe and create a project in the `%SOAM_HOME%\4.0\samples\CrossLanguage\COM\output` directory.

Package the service

Select the service in the language of your preference to deploy.

Important:

Deploy the service from only one language. This is because all services are registered under the same application, so only one service can be deployed at any one time.

Instructions are provided for all programming languages.

Package the C++ sample service

On Windows

To run the service, you first need to create a service package.

1. Go to the directory in which the compiled samples are located.

   cd %SOAM_HOME%\5.1\samples\CrossLanguage\CPP\Output

2. Create the service package by compressing the service executable into a zip file.

   gzip CrossLanguageServiceCPP.exe

   You have now created your service package `CrossLanguageServiceCPP.exe.gz`.

On Linux

To run the service, you first need to create a service package, then deploy it.

1. Change to the directory in which the compiled samples are located:

   cd $SOAM_HOME/5.1/samples/CrossLanguage/CPP/Output

2. Create the service package:

   tar -cvf CrossLanguageServiceCPP.tar CrossLanguageServiceCPP

   gzip CrossLanguageServiceCPP.tar

   You have now created your service package `CrossLanguageServiceCPP.tar.gz`.

Package the Java sample service

You must package the files required by your service to create a service package. When you built the sample, the service package was automatically created for you.

1. Go to the directory in which the service package is located.

   - On Windows, you have `CrossLanguageServiceJavaPackage.jar`

   cd %SOAM_HOME%\5.1\samples\CrossLanguage\Java

   - On Linux, you have `CrossLanguageServiceJavaPackage.zip`

   cd $SOAM_HOME/5.1/samples/CrossLanguage/Java

Package the .NET sample service

You must package the files required by your service to create a service package.

1. Go to the directory that contains the files for the service package:

   cd %SOAM_HOME%\5.1\samples\CrossLanguage\DotNet\CS\output

2. Locate the `CrossLanguageServiceDotNetCS.exe` and `Common.dll` files and add them to an archive using a compression program such as WinZip®.
3. Save the archive as `CrossLanguageServiceDotNetCS.zip` in the current directory.

Add the application

When you add an application through the DE PMC, you must use the Add Application wizard. This wizard defines a consumer location to associate with your application, deploys your service package, and

registers your application. After completing the steps with the wizard, your application should be ready to use.

1. In the DE PMC, click Symphony Workload > Configure Applications.

   The Applications page displays.
2. Select Global Actions > Add/Remove Applications.

   The Add/Remove Application page displays.
3. Select Add an application, then click Continue.

   The Adding an Application page displays.
4. Select Use existing profile and add application wizard. Click Browse and navigate to your application profile.
5. Select your application profile xml file, then click Continue

   Select the profile that matches the programming language and operating system for your service. For example, if you want to use a C++ service with a .NET and Java client, use the C++ application profile.

   - C++:
     - Windows—`%SOAM_HOME%\5.1\samples\CrossLanguage\CrossLanguageCpp.xml`
     - Linux—`$SOAM_HOME/5.1/samples/CrossLanguage/CrossLanguageCpp.xml`
   - Java
     - Windows—`%SOAM_HOME%\5.1\samples\CrossLanguage\crossLanguageJava.xml`
     - Linux—`$SOAM_HOME/5.1/samples/CrossLanguage/CrossLanguageJava.xml`
   - .NET:
     - `%SOAM_HOME%\5.1\samples\CrossLanguage\CrossLanguageDotNetCS.xml`

   The Service Package location window displays.
6. Browse to the service package you created in .zip, tar.gz, or .jar format and select it, then, select Continue.

   The Confirmation window displays.
7. Review your selections, then click Confirm.

   The window displays indicating progress. Your application is ready to use.
8. Click Close.

   The window closes and you are now back in the Platform Management Console. Your new application is displayed as enabled.

Run the sample clients

On Windows

To demonstrate that your cross-language clients and services work, use clients developed in different languages, and submit workload to a service developed in a different language.

We are going to use C++, Java, COM, and .NET clients to submit work to the service.

1. Run the C++ client on the command-line.

   %SOAM_HOME%\5.1\samples\CrossLanguage\CPP\Output\CrossLanguageClient

   You should see output on the command line as work is submitted to the system.

The client starts and the system starts the corresponding service. The client displays messages indicating that it is running.

2. Run the Java client on the command-line

```
%SOAM_HOME%\5.1\samples\CrossLanguage\Java\RunCrossLanguageClient.bat
```

You should see output on the command line as work is submitted to the system.

The client starts and the system starts the corresponding service. The client displays messages indicating that it is running.

3. Run the .NET client on the command-line.

```
%SOAM_HOME%\5.1\samples\CrossLanguage\DotNet\CS\output\CrossLanguageClient
```

The client starts and the system starts the corresponding service. The client displays messages in the text box indicating that it is running.

4. Run the COM client on the Synchronous Symphony Client window.

Note:

Make sure you have local administrator privileges to register the COM API assembly. Register Platform.Symphony.Soam.COM.dll, which is at the location %SOAM_HOME%\4.0\win32-vc7\lib\COM or %SOAM_HOME%\4.0\w2k3_x64-vc7-psdk\lib\COM with regsvr32. For example, regsvr32 Platform.Symphony.Soam.COM.dll.

```
%SOAM_HOME%\5.1\samples\CrossLanguage\COM\output\ComApiClients.exe
```

The client starts and the system starts the corresponding service. The client displays messages in the text box indicating that it is running.

On Linux

To demonstrate that your cross-language clients and services work, use clients developed in a different programming language from the service to submit workload.

For this example we are going to use C++ and Java clients to submit work to the service.

1. Run the C++ client application.

```
$SOAM_HOME/5.1/samples/CrossLanguage/CPP/output/CrossLanguageClient
```

You should see output as work is submitted to the system.

The client starts and the system starts the corresponding service. The client displays messages indicating that it is running.

2. Run the Java client application.

- From the command-line:

```
$SOAM_HOME/5.1/samples/CrossLanguage/Java/RunCrossLanguageClient.sh
```

You should see output as work is submitted to the system.

The client starts and the system starts the corresponding service. The client displays messages indicating that it is running.

Walk through the code

Review the sample code to learn how you can create cross-language clients and services.

Locate the code samples

Operating System	Language	File	Location of Code Sample
Windows	Java	Client	`%SOAM_HOME%\5.1\samples\CrossLanguage\Java\src\com\platform\symphony\samples\CrossLanguage\client\CrossLanguageClient.java`
		Input, output objects	`%SOAM_HOME%\5.1\samples\CrossLanguage\Java\src\com\platform\symphony\samples\CrossLanguage\common\MyMessage.java`
		Service	`%SOAM_HOME%\5.1\samples\CrossLanguage\Java\src\com\platform\symphony\samples\CrossLanguage\service\CrossLanguageService.java`
		Application profiles	The Java service and additional application parameters are specified in the application profile: `%SOAM_HOME%\5.1\samples\CrossLanguage\crossLanguageJava.xml`
		Output directory	`%SOAM_HOME%\5.1\samples\CrossLanguage\Java`
	C++	Client	`%SOAM_HOME%\5.1\samples\CrossLanguage\CPP\Client\CrossLanguageClient.cpp`
		Input, output objects	`%SOAM_HOME%\5.1\samples\CrossLanguage\CPP\Common\MyMessage.cpp`
		Service	`%SOAM_HOME%\5.1\samples\CrossLanguage\CPP\Service\CrossLanguageService.cpp`
		Application profiles	The C++ service and additional application parameters are specified in the application profile: `%SOAM_HOME%\5.1\samples\CrossLanguage\CrossLanguageCpp.xml`
		Output directory	`%SOAM_HOME%\5.1\samples\CrossLanguage\CPP\Output`

Operating System	Language	File	Location of Code Sample
	.NET	Client	`%SOAM_HOME%\5.1\samples\CrossLanguage\DotNet\CS\Client\CrossLanguageClient.cs`
		Input, output objects	`%SOAM_HOME%\5.1\samples\CrossLanguage\DotNet\CS\Common\MyMessage.cs`
		Service	`%SOAM_HOME%\5.1\samples\CrossLanguage\DotNet\CS\Service\CrossLanguageService.cs`
		Application profiles	The .NET service and additional application parameters are specified in the application profile: `%SOAM_HOME%\5.1\samples\CrossLanguage\CrossLanguageDotNetCS.xml`
		Output directory	`%SOAM_HOME%\5.1\samples\CrossLanguage\DotNet\CS\output`
	COM	Client	`%SOAM_HOME%\5.1\samples\CrossLanguage\COM\Client\VB_Clients.vbp`
		Input, output objects	`MyMessage.cls` is under Class Modules in VB.
		Output directory	`%SOAM_HOME%\5.1\samples\CrossLanguage\COM\output`
Linux	Java	Client	`$SOAM_HOME/5.1/samples/CrossLanguage/Java/src/com/platform/symphony/samples/CrossLanguage/client/CrossLanguageClient.java`
		Input, output objects	`$SOAM_HOME/5.1/samples/CrossLanguage/Java/src/com/platform/symphony/samples/CrossLanguage/common/MyMessage.java`
		Service	`$SOAM_HOME/5.1/samples/CrossLanguage/Java/src/com/platform/symphony/samples/CrossLanguage/service/CrossLanguageService.java`
		Application profiles	The Java service and additional application parameters are specified in the application profile: `$SOAM_HOME/5.1/samples/CrossLanguage/CrossLanguageJava.xml`
		Output directory	`$SOAM_HOME/5.1/samples/CrossLanguage/Java`

Operating System	Language	File	Location of Code Sample
	C++	Client	`$SOAM_HOME/5.1/samples/CrossLanguage/CPP/Client/CrossLanguageClient.cpp`
		Input, output objects	`$SOAM_HOME/5.1/samples/CrossLanguage/CPP/Common/MyMessage.cpp`
		Service	`$SOAM_HOME/5.1/samples/CrossLanguage/CPP/Service/CrossLanguageService.cpp`
		Application profiles	The C++ service and additional application parameters are specified in the application profile: `$SOAM_HOME/5.1/samples/CrossLanguage/CrossLanguageCpp.xml`
		Output directory	`$SOAM_HOME/5.1/samples/CrossLanguage/CPP/Output`

What the samples do

The client application sends 10 input messages with the data "Hello Grid !!" through Symphony to the service.

The service takes input data sent by the client application and returns the input and the reply "Hello Client !!". The client blocks to receive messages synchronously.

Differences between cross-language and same-language clients and services

Client and service structures are the same in cross-language clients and services as those of same-language clients and services, except for serialization.

The Java and .NET APIs support two modes of serialization: native serialization and Symphony serialization. The same-language samples for both of these APIs demonstrate the use of native serialization. On the other hand, the C++ and cross-language samples implement Symphony serialization.

Symphony serialization

Symphony serialization allows communication between clients and services written in different languages. For example, Symphony serialization allows you to use a C++ client with a Java service.

Symphony serialization is achieved in all languages by deriving from the SOAM Message object and implementing the appropriate serialization handlers.

You also use Symphony serialization if you are concerned with performance and memory usage.

Compatibility matrix

The following compatibility matrix for cross-language support shows which data types are compatible across languages.

C++	Java	.NET	VB	bits
short	short	Int16	Short	16

C++	Java	.NET	VB	bits
int	int	Int32	Integer	32
long long	long	Int64	Long	64
unsigned short		UInt16	UShort	16
unsigned int		UInt32	UInteger	32
unsigned long long		UInt64	ULong	64
float	float	Single	Single	32
double	double	Double	Double	64
char	char	Char	Char	-
bool	boolean	Boolean	Boolean	8
const char*	java.lang.String	String	String	N/A

Note:

- In Java and .NET, a character may consume 1-2 bytes of memory while in C++ a character consumes 1 byte of memory.
- Since byte arrays are represented differently across the supported languages, you need to use a special method when serializing this data. To write a byte array in C++ and Java, use the writeByteArray() method on the OutputStream. To write a byte array in .NET, use the WriteByteArray() method on the OutputStream.
- Note that the C++ “long” type and "unsigned long type" has been removed from the Compatibility Matrix for Symphony releases 3.1 and later due to portability issues across platforms and languages. While the C++ long type can still be serialized/de-serialized in the API, developers should consider using “int” for 32-bit values and “long long” for 64-bit values to maintain platform independence. Refer to the section on 64-bit Application Support for more information.

Use Symphony serialization in C++ to serialize input and output

There is no native serialization in C++. All C++ tutorials already use Symphony serialization. The client and service have the same structure as samples in all C++ tutorials.

Use Symphony serialization in Java to serialize input and output

Send input to the service

In the client, in `CrossLanguageClient.java`, when sending input to the service, session.sendTaskInput() takes in the message object instead of java.io.Serializable.

```
 ...
// Now we will send some messages to our service
                    int tasksToSend = 10;
                    for (int taskCount = 0; taskCount < tasksToSend; taskCount++)
                    {
                        // Create a message
                        MyMessage inMsg = new MyMessage(taskCount, true, "Hello Grid !!");
                        // send it
                        TaskInputHandle input = session.sendTaskInput(inMsg);
 ...
```

Serialize input

In `MyMessage.java`, implement the onSerialize() handler to write data to the provided OutputStream. Symphony calls this method when you send data.

In native serialization, the Java serialization mechanisms automatically serialize your data. For Symphony serialization, you need to specify the data you want to serialize.

Note:

Anything that you write to the stream you need to read back in the same order that you wrote it.

```
...
 public void onSerialize(OutputStream stream) throws SoamException
    {
        stream.writeInt(m_int);
        stream.writeBoolean(m_isSync);
        stream.writeString(m_string);
    }
...
```

Retrieve input on the service

In `CrossLanguageService.java`, create an instance of the message object and pass your own instance to populate the inMsg message object. The populateTaskInput() method fills in the object.

```
...
public void onInvoke (TaskContext taskContext) throws SoamException
    {
        // Get the input that was sent from the client
        MyMessage inMsg = new MyMessage();
        taskContext.populateTaskInput(inMsg);
...
```

In `MyMessage.java`, implement the onDeSerialize() handler to read data from the provided InputStream. Symphony calls this method when you retrieve data.In native serialization, the Java serialization mechanisms automatically deserialize your data. For Symphony serialization, you need to specify the data to read from the stream.

Note:

Anything that you write to the stream you need to read back in the same order that you wrote it.

```
...
public void onDeserialize(InputStream stream) throws SoamException
    {
        m_int = stream.readInt();
        m_isSync = stream.readBoolean();
        m_string = stream.readString();
    }
...
```

Send output back to the client

In `CrossLanguageService.java`, pass the output message object to send output back to the client. Symphony invokes your onSerialize() handler to send the output back to the client.

```
...
// Set our output message
        taskContext.setTaskOutput(outMsg);
...
```

Retrieve output on the client

In your client, in `CrossLanguageClient.java`, create an instance of the message object and pass your own instance to populate the outMsg object. The populateTaskOutput() method fills in the object.

Symphony invokes your onDeserialize() handler to retrieve the output.

```
...
// get the message returned from the service
                            MyMessage outMsg = new MyMessage();
                            output.populateTaskOutput(outMsg);
...
```

Use Symphony serialization in .NET to serialize input and output

Send input to the service

In your client, in `CrossLanguageClient.cs`, when sending input to the service, session.SendTaskInput () takes a message object instead of a [serializable] object.

```
 ...
// Now we will send some messages to our service
                    int numTasksToSend = 10;
                    for (int taskCount = 0; taskCount < numTasksToSend; taskCount++)
                    {
                        // Create a message
                      MyMessage inputMessage = new MyMessage(taskCount, true, "Hello Grid !!");
                        // send it
                        TaskInputHandle input = session.SendTaskInput(inputMessage);
 ...
```

Serialize input

In `MyMessage.cs`, implement the OnSerialize() handler to write data to the provided OutputStream. Symphony calls this method when you send data.

In native serialization, .NET serialization mechanisms automatically serialize your data. For Symphony serialization, you need to specify the data you want to serialize.

Note:

Anything that you write to the stream you need to read back in the same order that you wrote it.

```
...
public override void OnSerialize(OutputStream ostream)
        {
            ostream.WriteInt32(m_id);
            ostream.WriteBoolean(m_isSync);
            ostream.WriteString(m_string);
        }
...
```

Retrieve input on the service

In `CrossLanguageService.cs`, create an instance of the message object and pass your own instance to populate the inputMsg object. The PopulateTaskInput() method fills in the object.

```
...
public override void OnInvoke(TaskContext taskContext)
{
// get the input that was sent from the client
MyMessage inputMsg = new MyMessage();
taskContext.PopulateTaskInput(inputMsg);
...
```

In `MyMessage.cs`, implement the OnDeserialize() handler to read data from the provided InputStream. Symphony calls this method when you retrieve data.

In native serialization, .NET serialization mechanisms automatically deserialize your data. For Symphony serialization, you need to specify the data to read from the stream.

Note:

Anything that you write to the stream you need to read back in the same order that you wrote it.

```
...
public override void OnDeserialize(InputStream istream)
        {
            m_id     = istream.ReadInt32();
            m_isSync = istream.ReadBoolean();
            m_string = istream.ReadString();
        }
...
```

Send output back to the client

In `CrossLanguageService.cs`, pass the output message object to send output back to the client. Symphony invokes your OnSerialize() handler to send the output back to the client.

```
...
// set our output message
taskContext.SetTaskOutput(outputMsg);
...
```

Retrieve output on the client

In your client, in `CrossLanguageClient.cs`, create an instance of the message object and pass your own instance to populate the outputMessage object. The PopulateTaskOutput() method fills in the object. Symphony invokes your OnDeSerialize() handler to retrieve the output.

```
...
 // get the message returned from the service
                            MyMessage outputMessage = new MyMessage();
                            output.PopulateTaskOutput(outputMessage);
...
```

COM Tutorial

Tutorial: Developing a COM API client

Scope

Symphony COM API supports synchronous message input and output.

Symphony COM API is intended for clients written with VB6.0, VB Script, or VBA, and services written with C++, Java, or .NET.

Goal

This tutorial walks you through the sample client application code, then guides you through the process of building, packaging, and deploying the associated service.

This tutorial is based on the VBA client code in the `SoamExcelSample.xls` spreadsheet. The sample code demonstrates how to connect to an application using the COM API.

To help grasp the concepts, the spreadsheet actually contains two code samples. The first sample shows how a simple calculation is performed locally within the spreadsheet itself. The second sample extends the first sample to a grid-ready version by implementing the computational logic as a C++ service that can run on compute hosts in a Symphony cluster.

At a glance

1. Build the sample client and service
2. Package the sample service
3. Add the application
4. Run the sample client and service
5. Walk through the code

Prerequisites

- Ensure you have installed and started Symphony Developer Edition.

Where to find the documentation

Note:

%SOAM_HOME% is an environment variable that represents the Symphony DE installation directory; for example, `C:\SymphonyDE\DE51`.

Additional documentation is included in the *%SOAM_HOME%*`\docs` directory, as follows:

- COM API Reference: *%SOAM_HOME%*`\docs\symphonyde\5.1\com\api_reference`
- Platform Symphony Reference: *%SOAM_HOME%*`\docs\symphonyde\5.1\reference_sym`
- Error Reference: *%SOAM_HOME%*`\docs\symphonyde\5.1\error_reference`
- Platform Symphony DE Knowledge Center: *%SOAM_HOME%*`\docs\symphonyde\5.1\index.html`

Build the sample service

1. Open the Visual C++ solution file `ExcelSampleService_vc71.sln` from the location `%SOAM_HOME%\5.1\samples\COM\service`
2. Build the solution by pressing ctrl+ shift+B.

   Compiled executables and libraries are in the `%SOAM_HOME%\5.1\samples\COM\output` directory.

Package the sample service

You must package the files required by your service to create a service package.

1. Go to the directory that contains the file for the service package:

   cd %SOAM_HOME%\5.1\samples\COM\output
2. Locate the ExcelSampleService.exe file. Add the file to an archive using a compression program such as gzip. Save the archive as ExcelSampleService.exe.gz in the current directory.

Add the application

When you add an application through the DE PMC, you must use the Add Application wizard. This wizard defines a consumer location to associate with your application, deploys your service package, and registers your application. After completing the steps with the wizard, your application should be ready to use.

1. Click Symphony Workload > Configure Applications.

   The Applications page displays.
2. Select Global Actions > Add/Remove Applications.

   The Add/Remove Application page displays.
3. Select Add an application, then click Continue.

   The Adding an Application page displays.
4. Select Use existing profile and add application wizard, and browse to your application profile.
5. Select your application profile xml file, then click Continue.

   For ExcelSample, you can find your profile in the following location:

   - Windows—`%SOAM_HOME%\5.1\samples\COM\ExcelSample.xml`

   The Service Package location window displays.
6. Browse to the service package you created in .zip or .gz format and select it, then select Continue.

   The Confirmation window displays.
7. Review your selections, then click Confirm.

   The window displays indicating progress. Your application is ready to use.
8. Click Close.

   The window closes and you are now back in the Platform Management Console. Your new application is displayed as enabled.

Run the sample client and service

To run the service, run the client application. The service that a client application uses is specified in the application profile.

Note:

Make sure you have local administrator privileges to register the COM API assembly. Before running the service, log on as local administrator to register COM API.

1. Register `Platform.Symphony.Soam.COM.dll`, which is at the location `%SOAM_HOME%\5.1\win32-vc7\lib\COM` or `%SOAM_HOME%\5.1\w2k3_x64-vc7-psdk\lib\COM` with regsvr32:

   ```
   regsvr32 Platform.Symphony.Soam.COM.dll
   ```

   Note:

   For Windows Vista, right-click **Command Prompt** and select **Run as Administrator** before entering the command.

2. Go to the directory that contains `SoamExcelSample.xls`

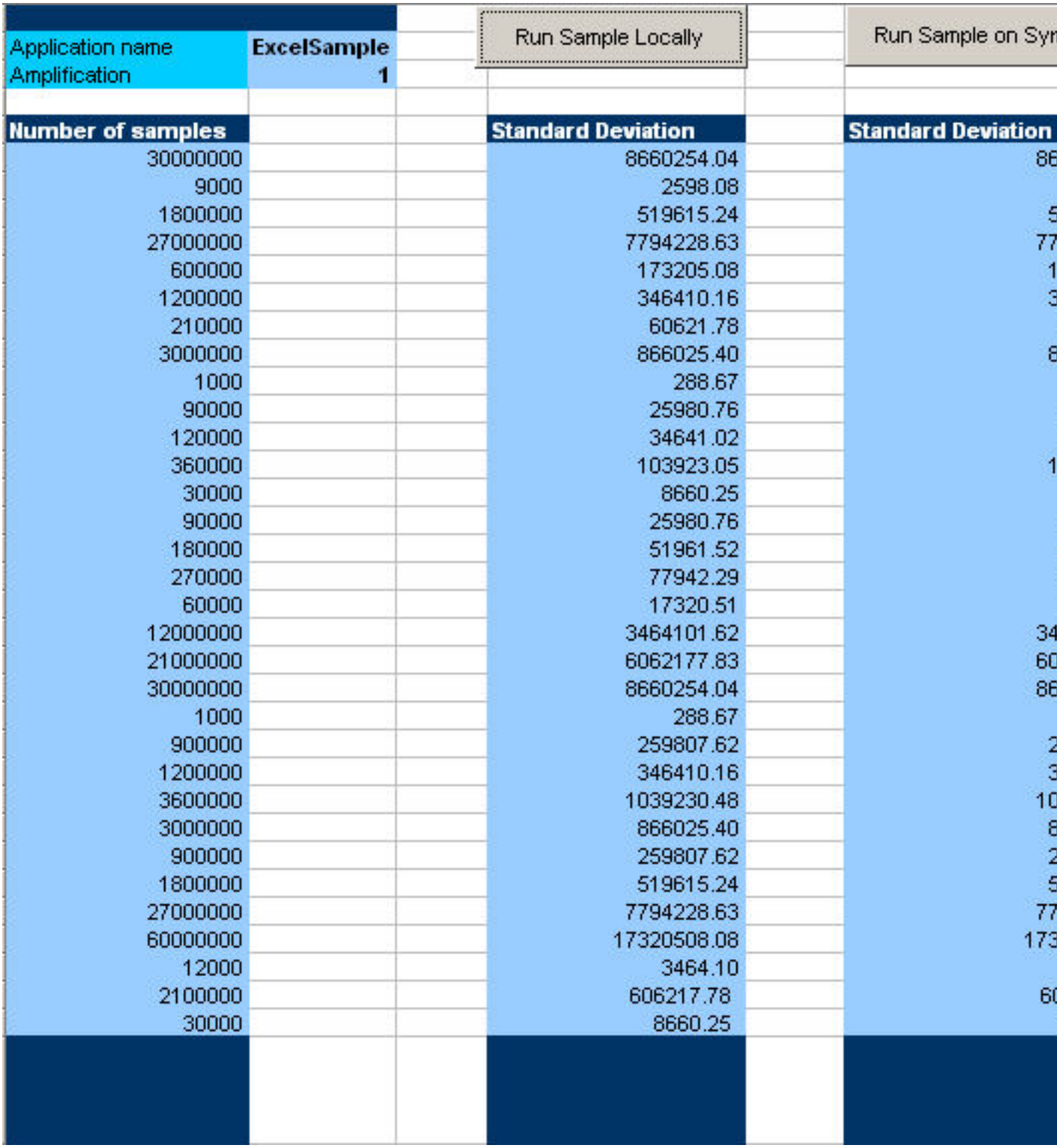
   ```
   %SOAM_HOME%\5.1\samples\COM\client
   ```

3. In the `SoamExcelSample.xls` spreadsheet, click Tools>Macro>Visual Basic Editor or press **Alt +F11**.
4. In the Visual Basic window, click Tools>References and select Platform.Symphony.Soam.COM 1.0 Type Library.
5. Click Browse and open `Platform.Symphony.Soam.COM.dll` from the location `%SOAM_HOME%\5.1\win32-vc7\lib\COM` or `%SOAM_HOME%\5.1\w2k3_x64-vc7-psdk\lib\COM`. After clicking OK on the References window, close the Visual Basic window.
6. On the SoamExcelSample.xls spreadsheet, click the Run Sample on Symphony button to run the application.

   Note:

   If you are using Excel 2000, make sure you set the Excel Macro security level to medium or low.

   The client starts and the system starts the corresponding service. The client displays results in the spreadsheet indicating that it is running.



Application name	ExcelSample		Run Sample Locally		Run Sample on Syn
Amplification	1				
Number of samples			Standard Deviation		Standard Deviation
30000000			8660254.04		86
9000			2598.08		
1800000			519615.24		5
27000000			7794228.63		77
600000			173205.08		1
1200000			346410.16		3
210000			60621.78		
3000000			866025.40		8
1000			288.67		
90000			25980.76		
120000			34641.02		
360000			103923.05		1
30000			8660.25		
90000			25980.76		
180000			51961.52		
270000			77942.29		
60000			17320.51		
12000000			3464101.62		34
21000000			6062177.83		60
30000000			8660254.04		86
1000			288.67		
900000			259807.62		2
1200000			346410.16		3
3600000			1039230.48		10
3000000			866025.40		8
900000			259807.62		2
1800000			519615.24		5
27000000			7794228.63		77
60000000			17320508.08		173
12000			3464.10		
2100000			606217.78		60
30000			8660.25		

Walk through the code

You will review the sample client application code to learn how you can create a synchronous client application that makes calls to the Symphony COM API.

Locate the code samples

Client

```
%SOAM_HOME%\5.1\samples\COM\client\SoamExcelSample.xls
```

Input/output object

```
%SOAM_HOME%\5.1\samples\COM\service\MyMessage.cpp
```

Service

```
%SOAM_HOME%\5.1\samples\COM\service\ExcelSampleService_vc71.sln
```

Application profile

The service and application parameters are defined in the application profile:

```
%SOAM_HOME%\5.1\samples\COM\ExcelSample.xml
```

What the samples do

The first sample implements computational logic that is processed locally in the spreadsheet. It calculates the standard deviation of 32 sets of values and populates the spreadsheet with the results.

The second sample features a synchronous client that sends 32 input messages to a Symphony service via the COM API. The service takes the input data, performs the calculations, and returns the results.

Since both samples implement the same computational logic, the results are identical.

Local sample

This sample executes locally on the Excel spreadsheet and all the code is contained within the spreadsheet.

Step 1: Get the input data

The `CommandButton2_Click` event encapsulates the client logic. Follow these steps to locate the `CommandButton2_Click` event code in the spreadsheet:

1. Select Tools > Macro > Visual Basic Editor.
2. In the Project Explorer, double-click Sheet1.
3. In the Object list box, select CommandButton2.

The *taskToSend* variable represents the number of input values for the standard deviation algorithm.

We initialize a range of cells where the results will be displayed on the spreadsheet. Next, we declare and initialize a two-dimensional array (values) to hold the results. The array indices correspond to the rows and columns for displaying the results on the spreadsheet.

A for loop increments the row index of the array and cycles through the inputs for the standard deviation calculations, which are read from the spreadsheet. An amplification factor is included in the input to the `StandardDeviation()` function to increase the input value, if required. The return value of the function is assigned to the array which, in turn, is assigned to the range object that displays the results in the spreadsheet cells.

```
...
Private Sub CommandButton2_Click()
    Dim k As Integer
    Dim r As Range

    Dim taskToSend As Long
    taskToSend = 32

    Dim amplification As Integer
    amplification = Range("B8").Value

    Set r = Range("D11", "D42")
    Dim values(0 To 32, 0 To 1)

    'Cleanup the cells
  For k = 0 To taskToSend - 1
    values(k, 0)=""
    Next k
    r.value2 = values

    For k = 0 To taskToSend - 1
        Dim numberOfSamples As Double
        numberOfSamples = CStr(Range("A" & (k + 11)).Value)

        values(k, 0) = StandardDeviation(numberOfSamples *
        amplification)
        r.value2 = values
    Next k
End Sub
...
```

Step 2: Implement the computational logic

The `StandardDeviation()` function contains the algorithm for calculating the standard deviation. The data set that the algorithm works on is derived from each input value. The result is passed back to the `CommandButton2_Click` event code and displayed on the spreadsheet.

```
...
Private Function StandardDeviation(ByVal numberOfSamples As Double)
    Dim i As Long
  Dim mean As Double

    StandardDeviation = 0
    mean = 0

    For i = 0 To numberOfSamples - 1
        mean = mean + i
    Next i

    mean = mean / numberOfSamples

    For i = 0 To numberOfSamples - 1
    StandardDeviation = StandardDeviation + (i - mean) * (i -
        mean)
  Next i

    StandardDeviation = StandardDeviation / numberOfSamples
    StandardDeviation = Sqr(StandardDeviation)
End Function
...
```

Symphony sample

This sample invokes the COM API to access a Symphony service.

Step 1: Connect to the application

The `CommandButton1_Click` event encapsulates the client logic. Follow these steps to locate the `CommandButton1_Click` event code in the spreadsheet:

1. Select Tools > Macro > Visual Basic Editor.
2. In the Project Explorer, double-click Sheet1.
3. In the Object list box, select CommandButton1.

To send data to a service for processing, you must first connect to an application. You specify an application name, a user name, and password. The application name must match the one defined in the application profile. In this sample, the application name is read from the spreadsheet.

For Symphony Developer Edition, there is no security checking and login credentials are ignored—you can specify any user name and password, such as "Guest". Security checking is done however, when your client application submits workload to the actual grid. The default security callback encapsulates the callback for the user name and password.

```
...
Private Sub CommandButton1_Click()
    On Error GoTo ReturnFailure
    Dim connection As CSoamConnection
    Range("F43").Value="Test Started...."

    'Initialize the Soam context
    Set soamApi = New CSoamAPI
    soamApi.Initialize

    'Get the Symphony application name from the spreadsheet
    Dim AppName As String
    AppName = CStr(Range("B7").Value)

   'Provide the credentials for the grid
  Dim callback As IDefaultConnectionSecurityCallback
    Set callback = New CDefaultConnectionSecurityCallback
    Call callback.Init("Guest", "Guest")

    'Connect to the grid
    Set connection = soamApi.Connect(AppName, callback)

    Range("F43").Value="Connection ID " & connection.Id
...
```

Step 2: Create a session to group tasks

A session is a way of logically grouping tasks that are sent to a service for execution. In this sample, the tasks are sent and received synchronously.

When creating a session, you need to specify the session attributes by using the `CSoamSessionCreationAttributes` object. We create a `CSoamSessionCreationAttributes` object called attributes and set three parameters in the object.

```
                    ...
  'Set the session attributes
  Dim attributes As CSoamSessionCreationAttributes
  Set attributes = New CSoamSessionCreationAttributes
  attributes.SessionName="ShortRunningTasks"
  attributes.SessionType="ShortRunningTasks"
  attributes.SessionFlags = SessionFlags.ReceiveSync

  'Create a session on the grid
  Dim session As CSoamSession
  Set session = connection.CreateSession(attributes)
  Range("F43").Value="Session created. Session ID " & session.Id
...
```

In this example, note that:

- The first parameter is the session name. This is optional. It can be any descriptive name you want to assign to your session. It is for information purposes.
- The second parameter is the session type. The session type is optional. You can leave this parameter blank and system default values are used for your session.

  Important:

  The session type must be the same session type as defined in your application profile.

  In the application profile, you define characteristics for the session with the session type.
- The third parameter is the session flag. When creating a synchronous session, set the flag to `SessionFlags.ReceiveSync`. This flag indicates to Symphony that this is a synchronous session.

Step 3: Send input data to be processed

The *taskToSend* variable represents the number of individual data sets (messages) that will be sent to the service. Next, we initialize a range of 32 cells where the results will be displayed on the spreadsheet. We declare and initialize a two-dimensional array (values) to hold the results. The array indices correspond to the rows and columns for displaying the results on the spreadsheet.

The next step is to create the input messages to be processed by the service. We call the `MyMessage` constructor and pass four input parameters. Note the input parameters:

- The first parameter is the number of samples. This value is the input data for each standard deviation calculation performed by the service.
- The second parameter is the line number. This value represents the row in the spreadsheet. As each message is sent, the row value is incremented until all the messages are sent.
- The third parameter is the column number, which represents the column in the spreadsheet. This value is fixed at 0 since the results will be displayed in a single column. The column value is echoed back to the client in each output message from the service.
- The fourth parameter is an input message string. It is not used in this sample.

When a message is sent, a task input handle is returned. This task input handle contains the ID for the task that was created for this input message.

```
...
    Dim k As Integer
    Dim r As Range

    Dim taskToSend As Long
    taskToSend = 32

    Dim amplification As Integer
    amplification = Range("B8").Value

    Set r = Range("F11", "F42")
    Dim values(0 To 32, 0 To 1)

    'Cleanup the cells
  For k = 0 To taskToSend - 1
      values(k, 0)=""
    Next k
    r.value2 = values
    'Start calculations
    For k = 0 To taskToSend - 1
        Dim numberOfSamples As Double
        numberOfSamples = CStr(Range("A" & (k + 11)).Value)

        Dim message As MyMessage
        Set message = New MyMessage

      message.numberOfSamples = numberOfSamples * amplification
        message.line = k
        message.column = 0
        message.StringMessage=""
        Dim inputHandler As CSoamTaskInputHandle
        Set inputHandler = session.SendTaskInput(message)

      Range("F43").Value="Sent Message number " & k & "Task ID
        " & inputHandler.Id
    Next k
...
```

Step 4: Retrieve the output

For each message that is sent to the service, call the `FetchTaskOutput()` method to retrieve the output message that was produced by the service. By passing a "1" to the method, we are retrieving only one result at a time. Consequently, the return value is an enumeration containing only one completed task result. This was done for demonstration purposes only. Typically, you would pass a value to the `FetchTaskOutput()` method that represents the total number of tasks sent to the service.

Check that the output for each task was successful before using the `PopulateTaskOutput()` method to extract the message; otherwise an exception is thrown. Load the result into the array and assign the array to the range object so that the result can be displayed in the appropriate spreadsheet cell.

```
...
    Range("F43").Value="Waiting for results...."
    Debug.Print "Fetching results."

    For k = 0 To taskToSend - 1
       Dim taskEnum As CSoamEnum
       Set taskEnum = session.FetchTaskOutput(1)
       Range("F43").Value="Retrieved task " & taskEnum.Count
       Dim output As CSoamTaskOutputHandle
       For Each output In taskEnum

       Range("F43").Value="Retrieved task with ID " & output.Id
          If output.IsSuccessful Then

           Dim outMessage As MyMessage
           Set outMessage = New MyMessage
           Call output.PopulateTaskOutput(outMessage)
             Debug.Print "Retrieved message " &
             outMessage.StringMessage

             Dim i, j As Integer
             i = outMessage.line
             j = outMessage.column
               values(i, j) = outMessage.StringMessage
          Else

             Debug.Print output.Id & " Task failed."
           Dim exception As CSoamCOMException
             Set exception = output.GetException()

             Dim reason As String
             reason=""
             Call exception.What(reason)
             Debug.Print output.Id & " task failed: " & reason
           Range("F43").Value = output.Id & " task failed.
             Reason for failure: " & reason
           End If
     Next output
       r.value2 = values
    Next k
...
```

Step 5: Define a service container

In this sample, calculations on input data are performed by a program that is implemented as a service. A service can be deployed on numerous compute hosts and run as concurrent service instances. For a service to be managed by Symphony, it needs to be in a container object. This is the service container.

In `ExcelSampleService.cpp`, `MyServiceContainer` inherits from the base class `ServiceContainer`.

```
...
class MyServiceContainer : public ServiceContainer
{
...
```

Step 6: Process the input

The middleware triggers the invocation of the ServiceContainer's onInvoke() handler every time a task input is sent to the service to be processed. You must implement the onInvoke method to process the task input, perform your computation, and return the result of the computation to the client.

To gain access to the input message from the client, you call the `populateTaskInput()` method on the task context. The middleware is responsible for placing the input into the `taskContext` object.

The task context contains all information and functionality that is available to the service during an `onInvoke()` call in relation to the task that is being processed.

Create the output message and set the following message properties. The values for these properties are read from the input message and echoed back to the client.

- The first property is the input data for the calculation performed by the service.
- The second property is the row number for the spreadsheet.
- The third property is the column number for the spreadsheet.

Pass the input data (*m_number_of_samples*) to the `StandardDeviation()` method and format the output message string with the result. Passing the output message to the `SetTaskOutput()` method generates the task output message that is sent to the client.

```
...
virtual void onInvoke (TaskContextPtr& taskContext)
    {
      // get the input that was sent from the client
      MyMessage inMsg;
    taskContext->populateTaskInput(inMsg);
      // We simply echo the data back to the client
    MyMessage outMsg;
      outMsg.setnumberOfSamples(inMsg.getnumberOfSamples());
      outMsg.setLine(inMsg.getLine());
      outMsg.setColumn(inMsg.getColumn());
      m_numberOfSamples = inMsg.getnumberOfSamples();
      char buffer[128];
      sprintf(buffer, "%.2f",
      StandardDeviation(m_numberOfSamples));
      outMsg.setString(buffer);
      // set our output message
      taskContext->setTaskOutput(outMsg);
    }
...
```

Step 7: Run the container

The service is implemented within an executable. As a minimum, we need to create an instance of the service container within our main function and run it.

```
...
int main(int argc, char* argv[])
{
    // return value of our service program
    int retVal = 0;
    try
    {
        // Create the container and run it
        MyServiceContainer myContainer;
        myContainer.run();
    }
...
```

Step 8: Catch exceptions

Catch exceptions in case the container fails to start running.

```
...
    catch (SoamException& exp)
    {
        // report exception to stdout
        cout << "exception caught ... " << exp.what() << endl;
      retVal = -1;
    }
...
```

Step 9: Create the computational logic of the service

The computational logic of the service is implemented in the `StandardDeviation()` method. The method generates a data set of integers in the range of 0 up to the *numberOfSamples* value and applies a standard deviation algorithm to it.

```
...
 private:
  double StandardDeviation(double numberOfSamples)
  {
      double standardDeviation = 0;
      double mean = 0;

      for(int i = 0; i < numberOfSamples; i++)
      {
          mean = mean + i;
      }

      mean = mean / numberOfSamples;

    for(int i = 0; i < numberOfSamples; i++)
      {
        standardDeviation = standardDeviation + (i - mean) * (i
          - mean);
    }
      standardDeviation = standardDeviation / numberOfSamples;
      return sqrt(standardDeviation);
  }
...
```

Eclipse Tutorial

Tutorial: Developing a Symphony application with Eclipse

Goal

This tutorial walks you through the steps for developing application code using the Symphony plug-in for the Eclipse IDE. It then guides you through the process of building, packaging, and deploying the associated service. This tutorial was prepared for users that are already familiar with the Eclipse IDE.

Prerequisites

Note:

If you are running the Eclipse plug-in on a host that has a Symphony DE pre-5.1 version and Symphony DE 5.1 installed, you must ensure that the system environment is set up for Symphony DE 5.1. Refer to Installing Symphony Developer Edition in the Platform Knowledge Center.

Install the Symphony plug-in for Eclipse

For instructions on installing the Symphony plug-in into Eclipse, refer to the Symphony Plug-in for Eclipse topic in the Application Development Guide.

What the Symphony plug-in can do

The Symphony plug-in for Eclipse is an all-in-one tool package that eases the task of Symphony application development and package deployment.

The plug-in generates a framework of code that provides a foundation for Symphony application development. This framework of basic code is typically common to all Symphony applications. All you need to do is add your own logic to it. One of the key components of the plug-in is the Symphony project wizard, which guides you through the development process and prompts you for application-specific information.

The plug-in also facilitates service package deployment and application registration. This is achieved by combining multiple tasks into single operations through the Symphony DE Platform Management Console, which is integrated with the plug-in.

Where to find the documentation

You can access additional documentation such as the *Java API Reference*, the *Platform Symphony Reference*, and the *Error Reference* from the Symphony DE Knowledge Center.

Windows

- From the Start menu, select Programs > Platform Computing > Symphony Developer Edition 5.1.0 > Developer Knowledge Center

Linux

- `$SOAM_HOME/docs/symphonyde/5.1/index.html`

For convenience, the *Java API Reference* is also available via the Eclipse Help menu.

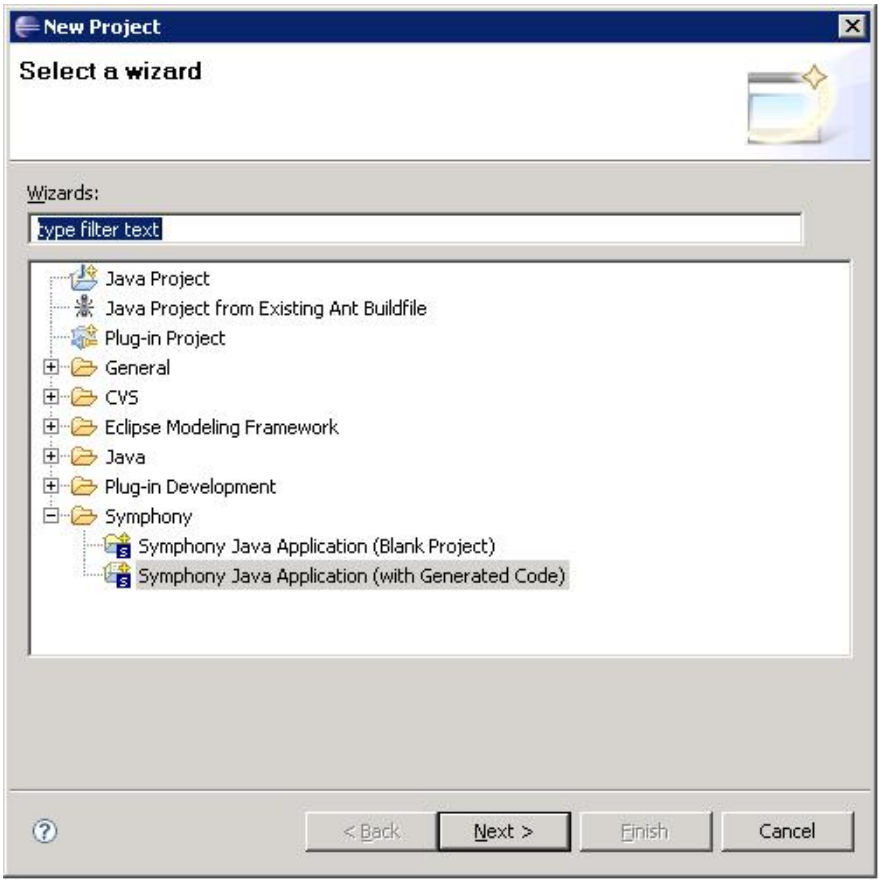
Create a new Symphony Java application

This section describes the steps for creating a new Symphony Java application (with generated code) using the Symphony project wizard in Eclipse.

Step 1 Create a new project

1. Select File > New > Project

   The New Project dialog appears.



2. Expand the Symphony wizard. Select Symphony Java Application (with Generated Code).
3. Click Next.

   The Symphony Application Identification dialog appears.

Step 2: Name the application and package

The application name is what binds the client to the service and it must be unique in the cluster.

1. Enter **MySymphonyApp** as the Symphony application name. Click Next.
2. Verify that the Create a package for the generated Java classes box is checked and enter the following package name:

   com.platform.symphony.foo

Note:

(Classes that are not placed in a named package belong to the "default package" associated with the current project directory.)

Step 3: Name the client and service classes

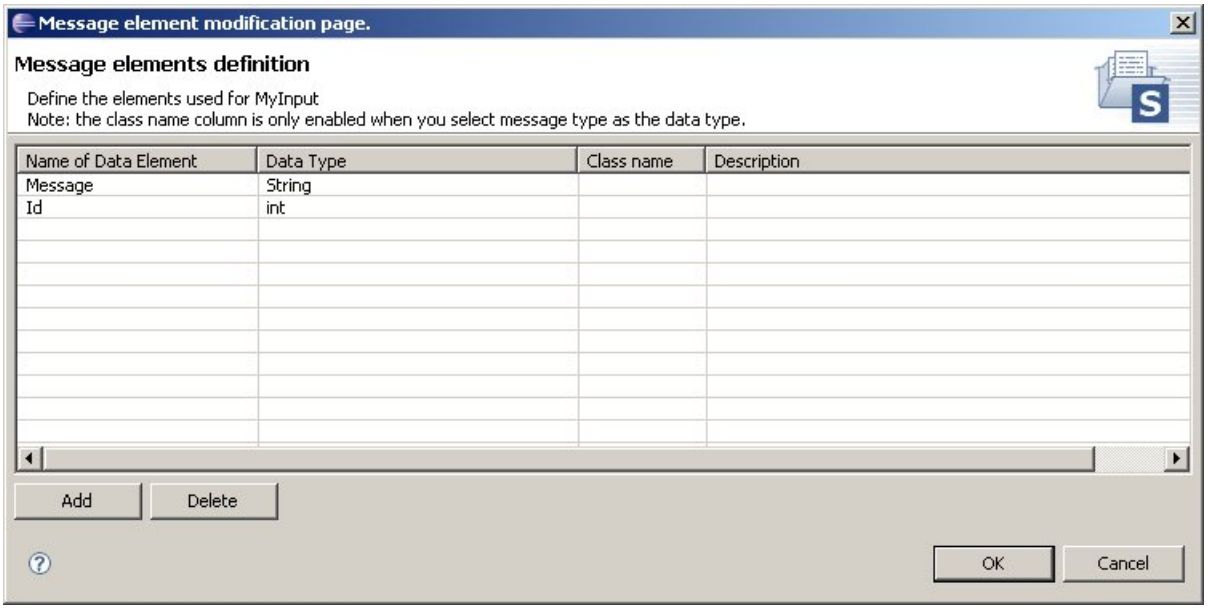
1. Enter **AsyncClient** as the client class name.
2. Select Async as the client type. An async client requires a callback class, which will be added to the project when the project is created.
3. Enter **MyService** as the service class name. Click Next.

Step 4: Define the message

Your client application needs to handle data that it sends as input, and output data that it receives from the service. You need to define message classes that implement methods to set and access the data, such as the message string and task ID.

1. In the New Message dialog, click Add.
2. Enter **MyInput** as the message class name to handle input messages.
3. Double-click Edit.

   The Message elements definition dialog appears.
4. Click Add. Enter **Id** as the name of the data element. Verify that the data type is int (integer).
5. Click Add. Enter **Message** as the name of the second data element. Set the data type to **String**.



6. Click OK.
7. In the New Message dialog, click Add.
8. Enter **MyOutput** as the message class name to handle output messages.
9. Repeat steps 3 to 5 for the MyOutput class..
10. Click Next.

    The Create a Java project dialog appears.

Step 5: Create the Java project

1. The project name can be any descriptive name you choose. For this tutorial, enter **MySymphonyProj** as the project name. Your project will be created in the workspace associated with Eclipse.
2. Select Create new project in workspace.
3. Select Use default JRE. Ensure that you are using JDK version 1.5.
4. Select Use project folder as root for sources and class files.
5. Click Finish.

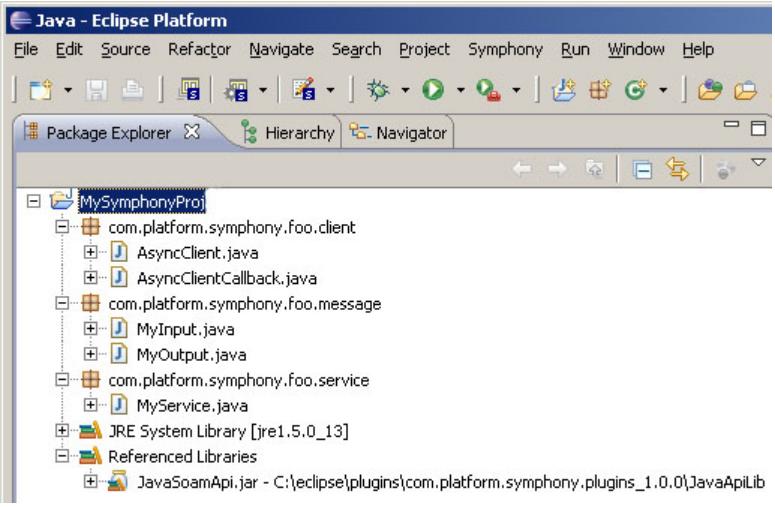
   Eclipse creates a new Java project in your workspace with subfolders for the newly-created classes. These classes contain generated code that can be used as a basis for Symphony application development.

   The Symphony plug-in for Eclipse also adds project-dependent files such as the JRE system library and Symphony API to the project.

Note:

If you only see the Eclipse welcome screen and not your project with the generated Java code, minimize the welcome screen.

To view your project in Package Explorer, select **Window** > **Show View** > **Package Explorer**.



Review and understand the generated code

We will review the client, message, and service code that is generated by the Symphony plug-in for Eclipse and discuss what you need to do to complete the application coding.

Client class

Import message class

Since you need to create an instance of the message class in your client class, you must import the message class. Add the following import statement to the generated code for the client class:

import com.platform.symphony.foo.message.*;

Connect to an application

A connection establishes a context for your client and workload. When you connect to an application:

- Application attributes defined in the application profile are used to provide context such as which service to use, session type, and any additional scheduling or application parameters.
- A connection object is returned.

The application name in the connection must match that defined in the application profile. This name was assigned when you created the new project using the wizard.

The default security callback encapsulates the callback for the user name and password. In Symphony DE, there is no security checking and login credentials are ignored —you can specify any user name and password. However, when using your client on the grid with Platform Symphony, you need a valid user name and password.

The generated code for connecting to the application is complete and does not require any additional code to make it functional.

Here is the generated code:

```
...
    // Set up application specific information to be supplied to Symphony
    String appName="MySymphonyApp";
    // Set up application authentication information using the default security provider

    DefaultSecurityCallback securityCB = new DefaultSecurityCallback("Guest","Guest");
    Connection connection = null;
    try
    {
      // Connect to the specified application
      connection = SoamFactory.connect(appName, securityCB);
      // Retrieve and print our connection ID
      System.out.println("connection ID=" + connection.getId());
...
```

```
    finally
    {
      // Mandatory connection close
      if (connection != null)
      {
          connection.close();
          System.out.println("Connection closed");
      }
    }
...
```

Important:

The creation and usage of the connection object is scoped in a try-finally block. The finally block, with the connection.close() method, ensures that the connection is always closed whether exceptional behavior occurs or not. Failure to close the connection causes the connection to continue to occupy system resources.

Create a session to group tasks

A session is a way of logically grouping tasks that are sent to a service for execution.

When creating a session, you need to specify the session attributes by using the SessionCreationAttributes object. The generated code sets four parameters in the SessionCreationAttributes object.

The first parameter is the session name. This is optional. The session name can be any descriptive name you want to assign to your session. It is for informational purposes, such as in the command line interface.

The second parameter is the session type. The session type is optional. You can leave this parameter blank and system default values are used for your session.

The third parameter is the session flag, which the code generator specified as Session.PARTIAL_ASYNC. This flag setting was determined by the client type specified in the project wizard. This indicates to Symphony that the client expects to receive messages asynchronously.

The fourth parameter is the callback object. This object is used by Symphony to call back to the client when the results are ready to be received.

The attributes object is passed to the createSession() method, which returns the created session.

Important:

Similar to the connection object, the creation and usage of the session (sending and receiving data) is scoped in a try-finally block. The finally block, with the session.close() method, ensures that the session is always closed, whether exceptional behavior occurs or not. Failure to close the session causes the session to continue to occupy system resources.

Here is the generated code:

```
...
    // Set up session attributes
    SessionCreationAttributes attributes = new SessionCreationAttributes();
    attributes.setSessionName("mySession");
    attributes.setSessionType("");  // we will use the default session type
    attributes.setSessionFlags(Session.PARTIAL_ASYNC);
    attributes.setSessionCallback(sessionCallback);

    // Create a asynchronous session
    Session session = null;
    try
    {
        session = connection.createSession(attributes);
        // Retrieve and print session ID
         System.out.println("Session ID:" + session.getId());
...
    finally
    {
        // Mandatory session close
        if (session != null)
        {
            session.close();
            System.out.println("Session closed");
        }
    }
...
```

Send input data to be processed

In this step, you create 10 input messages to be processed by the service. When a message is sent, a task input handle is returned. This task input handle contains the ID for the task that was created for this input message. The part of the code that is missing is shown by the TODO comments. To make this code functional, you must create an input message and attach it to the TaskSubmissionAttributes object, which is subsequently sent to the Symphony middleware.

Here is the generated code:

```
...
    // Now we will send some messages to our service
    for (int taskCount = 0; taskCount < tasksToSend; taskCount++)
    {
        // Create a message
        ////////////////////////////////////////////////////////////
        // TODO: Place code here to construct message object to be sent
        // eg . InputMessage myInput = new InputMessage(...)
        ////////////////////////////////////////////////////////////
        // Set task submission attributes
        TaskSubmissionAttributes taskAttr = new TaskSubmissionAttributes();
        ////////////////////////////////////////////////////////////
        // TODO: Place code here to set input for task submission
        // eg . taskAttr.setTaskInput(myInput);
        ////////////////////////////////////////////////////////////
        // Send it
        TaskInputHandle input = session.sendTaskInput(taskAttr);
        // Retrieve and print task ID
        System.out.println("task submitted with ID : " + input.getId());
    }
...
```

The following code snippet shows an example of the completed code for creating the message and attaching it to the TaskSubmissionAttributes object. In this case, we are inserting a simple string and task ID into the input message.

```
...
    // Now we will send some messages to our service
    for (int taskCount = 0; taskCount < tasksToSend; taskCount++)
    {
        // Create a message
        MyInput myInput = new MyInput();
      myInput.setId(taskCount);
    myInput.setMessage("Hello Grid!!");
        // Set task submission attributes
        TaskSubmissionAttributes taskAttr = new TaskSubmissionAttributes();
       taskAttr.setTaskInput(myInput);
        // Send it
        TaskInputHandle input = session.sendTaskInput(taskAttr);
        // Retrieve and print task ID
        System.out.println("task submitted with ID : " + input.getId());
    }
...
```

Synchronize the controlling and callback threads

This step is performed after sending the input data to be processed.

Since our client is asynchronous, we need to synchronize the controlling thread and the callback thread. In the generated code, the controlling thread blocks until all replies have come back.

The callback signals when all results are received.

Here is the generated code:

```
...
    synchronized(sessionCallback)
    {
        while (!sessionCallback.isDone())
        {
            sessionCallback.wait();
        }
    }
...
```

Message input/output classes

The MyInput class represents the data input to the service, and the MyOutput class represents the data output from the service. These classes implement methods to set and access the data, such as the message string and task ID.

The code that is generated uses Symphony's API to serialize the MyInput and MyOutput objects. Symphony serialization is achieved by deriving from the SOAM Message object and implementing the appropriate serialization handlers, i.e., onSerialize() and onDeserialize().

The following code was generated for the MyInput class:

```
...
public class MyInput extends com.platform.symphony.soam.Message
{
    //============================================
    //.Constructor.
    //============================================
    public MyInput()
    {
        super();
    }
    //============================================
    //.Accessors.
    //============================================
    // "Id" ()
    public int getId()
    {
        return this.Id;
    }
    public void setId(int value)
    {
        this.Id = value;
    }
    // "Message" ()
    public String getMessage()
    {
        return this.Message;
    }
    public void setMessage(String value)
    {
        this.Message = value;
    }
    //============================================
    //.Serialization - Deserialization.
    //============================================
    public void onSerialize(OutputStream ostream) throws SoamException
    {
        ostream.writeLong(this.V4350FA63DBAA4f65A190EDDE29709AC6);
        ostream.writeInt(this.Id);
        ostream.writeString(this.Message);
    }
    public void onDeserialize(InputStream istream) throws SoamException
    {
        this.V4350FA63DBAA4f65A190EDDE29709AC6 = istream.readLong();
        if (this.V4350FA63DBAA4f65A190EDDE29709AC6 != 1)
        {
            String errorMessage="A version mismatch error has occured.";
            errorMessage += "The message being deserialized is verion " +
            this.V4350FA63DBAA4f65A190EDDE29709AC6;
            errorMessage += " but we were expecting verion <1>.";
            errorMessage += "Verify that all message definitions are up to date.";
            throw new FatalException(errorMessage);
        }
        this.Id = istream.readInt();
        this.Message = istream.readString();
    }
...
```

Callback class

When the client type is set to asynchronous in the project wizard, it generates a callback class that extends the SessionCallback class.

Import message class

Since you need to create an instance of the message class in your callback class, you must import the message class. Add the following import statement to the generated code for the callback class:

import com.platform.symphony.foo.message.*;

Retrieve the output

This class contains the `onResponse()` method to retrieve the output for each input message that is sent. To make the generated code complete, you need to add code to the `onResponse()` method, as indicated by the TODO comments.

Note that:

- `onResponse()` is called every time a task completes and output is returned to the client. The task output handle allows the client code to manipulate the output.
- `isSuccessful()` checks whether there is output to retrieve.

```
public void onResponse(TaskOutputHandle output) throws SoamException
{
    try
    {
        // Check for success of task
        if (output.isSuccessful())
        {
            // Get the message returned from the service
            ////////////////////////////////////////////////////
            // TODO: Retrieve the result from the TaskOutputHandle
            // NOTE : If your output message was generated by the
            // Symphony Eclipse Pluggin
            // You must use the TaskOutputHandle.populateTaskOutput() method to
            // retrieve the output. The Symphony Eclipse
            // Pluggin generates messages inherited
            // from the "com.platform.symphony.soam.Message" class.
            // eg.
            // MyOutputMessage myOutput = new MyOutputMessage();
            // output.populateTaskOutput(myOutput);
            ////////////////////////////////////////////////////
            // Display content of reply
            ////////////////////////////////////////////////////
            // TODO: Display some reply
            // eg .
            // System.out.println("\nTask Succeeded [" +
            // output.getId() + "]");
            // System.out.println("Your Internal ID was : " +
            // myOutput.getId());
            // System.out.println("Estimated runtime was recorded
            // as : ");
            // System.out.println(myOutput.getRunTime());
            // System.out.println(myOutput.getString());
            ////////////////////////////////////////////////////
        }
        else
        {
            // Get the exception associated with this task
            SoamException ex = output.getException();
            System.out.println("Task Failed : ");
            System.out.println(ex.toString());
        }
    }
    catch (Exception exception)
    {
        System.out.println("Exception occured in onResponse() : ");
        System.out.println(exception.getMessage());
    }
    // Update counter used to synchronize the controlling thread
    // with this callback object
    incrementTaskCount();
}
```

The following code example shows the completed code for the `onResponse()` method. If there is output to retrieve, `populateTaskOutput()` gets the output. Once results return, print them to standard output and return.

```
public void onResponse(TaskOutputHandle output) throws SoamException
{
    try
    {
        // check for success of task
        if (output.isSuccessful())
        {
            // get the message returned from the service
            MyOutput myOutput = new MyOutput();
          output.populateTaskOutput(myOutput);
            // display content of reply
              System.out.println("\nTask Succeeded [" +  output.getId() + "]");
            System.out.println("Your Internal ID was : " + myOutput.getId());
            System.out.println(myOutput.getMessage());
        }
        else
        {
            // get the exception associated with this task
            SoamException ex = output.getException();
            System.out.println("Task Failed :");
            System.out.println(ex.getMessage());
        }
    }
    catch (Exception exception)
    {
        System.out.println("Exception occured in onResponse() : ");
        System.out.println(exception.getMessage());
    }
    // Update counter used to synchronize the controlling thread
    // with this callback object
    incrementTaskCount();
}
```

Service class

Import message class

Since you need to create an instance of the message class in your service class, you must import the message class. Add the following import statement to the generated code for the service class:

import com.platform.symphony.foo.message.*;

Process the input

Symphony calls `onInvoke()` on the service container once per task. This is where the calculation is performed.

Important:

Services are virtualized. As a result, a service should not read from stdin or write to stdout. Services can, however, read from and write to files that are accessible to all compute hosts.

To gain access to the data set from the client, you call the `populateTaskInput()` method on the taskContext. The Symphony middleware is responsible for placing the input into the taskContext object.

The task context contains all information and functionality that is available to the service during an `onInvoke()` call in relation to the task that is being processed.

The following code was generated by the Symphony plug-in. What is missing, as identified by the TODO comment, is your service logic for the `onInvoke()` method.

```
...
public void onInvoke (TaskContext taskContext) throws SoamException
{
    ////////////////////////////////////////////////////////////
    // TODO: Place your service logic here. This method will be
    //       called for each task that has to be processed.
    //
    // NOTE : If your message was generated by the Symphony Eclipse Pluggin
    // You must use the TaskContext.populateTaskInput() method to retrieve
    // the input. The Symphony Eclipse Pluggin generates messages inherited
    // from the "com.platform.symphony.soam.Message" class.
    // eg.
    //
    // // get input
    // MyInputMessage myInput = new MyInputMessage();
    // taskContext.populateTaskInput(myInput);
    //
    // // do some processing of input
    //  ...
    //
    // // set output (which is returned to client)
    // taskContext.setTaskOutput(myOutput);
    ////////////////////////////////////////////////////////////
}
...
```

The following code example shows the `onInvoke()` method with the service logic. The input message is simply echoed back to the client by creating a string in the output message and passing it to the `setTaskOutput()` method.

```
...
public void onInvoke (TaskContext taskContext) throws SoamException
{
    // We simply echo the data back to the client
    MyOutput myOutput = new MyOutput();
    // get the input that was sent from the client
    MyInput myInput = new MyInput();
    taskContext.populateTaskInput(myInput);
    // echo the ID
   myOutput.setId(myInput.getId());
    // setup a reply to the client
    StringBuffer sb = new StringBuffer();
    sb.append("Client sent : ");
    sb.append(myInput.getMessage());
    sb.append("\nSymphony replied : Hello Client !!");
   myOutput.setMessage(sb.toString());
    // set our output message
    taskContext.setTaskOutput(myOutput);
}
...
```

Create a deployment package

You must package the files required by your service to create a service package.

1. In the Eclipse Package Explorer, right-click on project MySymphonyProj. Select Symphony Operations > Create Deployment Package
2. Enter **MyService.zip** as the package name in the Specify the package name textbox.
3. Click Browse beside Select the package path to select the path where the service package will be stored.
4. Click Browse beside Specify Main Service Class Name. Enter **MyService** in the textbox. Click OK.

5. The JVM options can be used to configure JVM performance tuning such as memory allocation for the service instance. In this tutorial, we are using default JVM settings so it is not necessary to specify any JVM options.
6. Select `*.message.jar` and `*.service.jar` files in the Package name or file name list.
7. The option for Do stdout redirection from generated service script can be left unchecked. Selecting this option will cause standard output from the JVM to be redirected to a file located in the Symphony DE `work` directory. The file pattern is <main class name>_<computername>_<process ID of script>_<timestamp>.log.
8. Click Create and Validate Service Package.

   After creating and validating the service package, the following message is displayed:

   `Compressed` *`package_path`*`/MyService.zip successfully!`

   `The service package is valid and it is ready to be deployed to DE.`
9. Click Finish. The next step is to create an application profile and register your application.

Creating an application profile

The application profile defines the application behavior within Symphony and provides information that Symphony needs to run services and manage workload. It also binds the application to a specific service.

Note:

To create an application profile via the Symphony DE PMC, your Symphony DE cluster must be started.

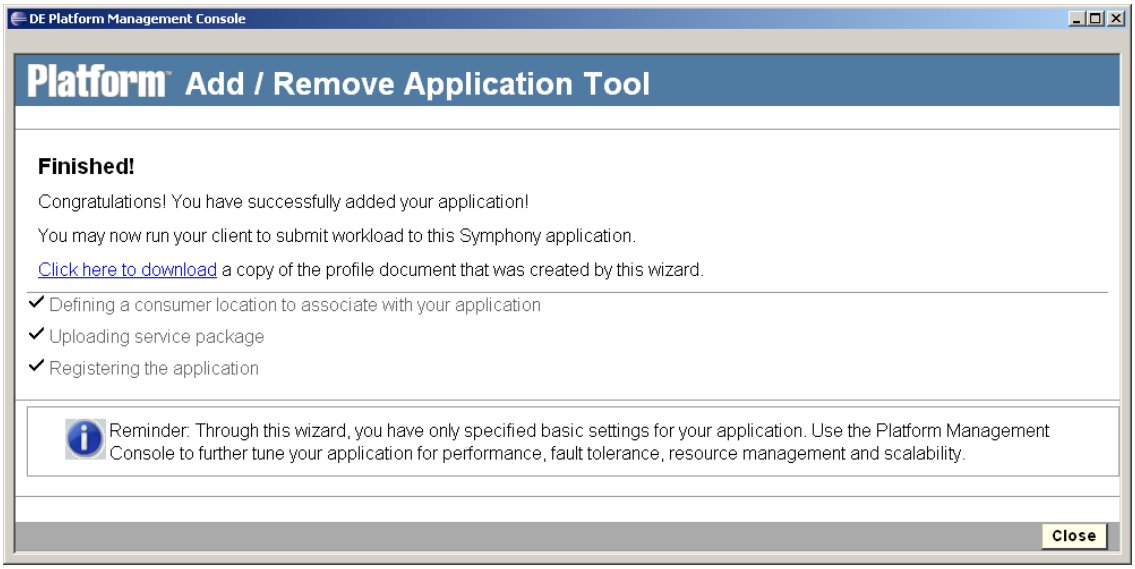
Create an application profile

1. In the Package Explorer, right-click on project MySymphonyProj. Select Symphony Operations > Add/Remove Application.
2. Select Add an application. Click Continue.
3. Select Create new profile and add application wizard. The wizard automatically populates the application name field with **MySymphonyApp**.

   Important:

   If your development environment has more than one version of JDK installed (for example, JDK versions 1.4 and 1.5), you should configure the PATH environment variable to point to JDK 1.5. Similarly, if you have 32-bit and 64-bit versions of JDK 1.5 installed, you should configure the PATH environment variable to point to the version that matches your 32-bit or 64-bit platform. Failure to configure the PATH correctly may prevent your service from running. To configure the PATH environment variable, click **Specify environment variables for this service**. Enter **PATH** as the Name and the path to the JDK 1.5 bin directory as the value.
4. Click Continue.
5. Click Browse and navigate to the location of the service package. Select the service package file `MyService.zip`. Click Continue.
6. Select System Defaults for the session type. Click Continue.
7. Click Confirm to accept the application profile definitions.

   The project wizard creates your application and registers it within Symphony.



Run the application

1. In the Package Explorer, right-click com.platform.symphony.foo.client class. Select Run As > Java Application. The application runs and prints results in the console window.

```
connection ID=df7f5956-ffff-ffff-c000-00096bb5e338-1824-728
Session ID:1
task submitted with ID : 1
task submitted with ID : 2
task submitted with ID : 3
task submitted with ID : 4
task submitted with ID : 5
task submitted with ID : 6
task submitted with ID : 7
task submitted with ID : 8
task submitted with ID : 9
task submitted with ID : 10

Task Succeeded [1]
Your Internal ID was : 0
Client sent : Hello Grid!!
Symphony replied : Hello Client !!

Task Succeeded [2]
Your Internal ID was : 1
Client sent : Hello Grid!!
Symphony replied : Hello Client !!

Task Succeeded [3]
Your Internal ID was : 2
Client sent : Hello Grid!!
Symphony replied : Hello Client !!

Task Succeeded [4]
Your Internal ID was : 3
Client sent : Hello Grid!!
Symphony replied : Hello Client !!

Task Succeeded [5]
```

2. To monitor the session status, right-click the project in the Package Explorer and select Symphony Operations > Monitor Workload.

   For more information about monitoring sessions and tasks, refer to the Symphony DE PMC help.

Service package re-deployment

You must update and re-deploy the service package if you have made modifications to the message or service code in your application.

1. In the Eclipse Package Explorer, right-click on your project. Select Symphony Operations > Create Deployment Package.
2. Verify the service information in the Symphony Service Packaging Utility dialog.
3. Click Create and Validate Service Package.

   After creating and validating the service package, the following message is displayed:

   `Compressed` *`package_path`*`/MyService.zip successfully!`

   `The service package is valid and it is ready to be deployed to DE.`
4. Click Deploy the Package.

Note:

The **Deploy the Package** button is only enabled when the application is registered. Clicking this button adds the service package to the Symphony DE repository.

After deploying the service package, the following message is displayed:

```
Service package was deployed successfully to DE.
```

5. Click Finish.

Importing samples into Eclipse

Every sample includes a .classpath file sufficient for building the project within Eclipse. Consequently, there is no need to import the project by importing the existing Ant build file.

Note:

Importing and execution of Java samples in Eclipse from Symphony DE TAR packages is not supported.

Note:

Importing samples into Eclipse from previous versions of Symphony DE (3.2 and lower) is not recommended. Older samples require reorganization of their directory structure and editing of the application profile.

Note:

You cannot pre-load information about message code that was generated outside the Eclipse plug-in since the code generation wizard is not aware of message code that has been created or modified externally. For example, if you define new message classes for an imported project using the code generation wizard, the existing message classes from the imported project will not appear in the Message Definition dialog.

In this section, we will import the `SampleApp` sample that is included with Symphony DE and deploy its service. The Java samples are located at:

Windows

- *`%SOAM_HOME%`*`\5.1\samples\Java`

Linux

- *`$SOAM_HOME`*`/5.1/samples/Java`

For more information about the `SampleApp` code sample or any other sample included with Symphony DE, refer to the appropriate tutorial or readme in the Knowledge Center.

Import samples into Eclipse

1. From the Eclipse menu, select File > Import.

   The Import dialog appears.
2. Double-click General. Double-click Existing Projects into Workspace.

3. Choose Select root directory. Click Browse. Browse to the `SampleApp` directory in Symphony DE. Click OK.
4. Click Finish.

Create and validate the service package

1. In the Eclipse Package Explorer, right-click on the sample project SampleApp. Select Symphony Operations > Create Deployment Package
2. Enter **SampleAppService.zip** as the package name in the Specify the package name textbox..
3. Click Browse to select the path where the service package will be stored.
4. Click Browse to specify the main service class name. Enter **MyService** in the textbox. Click OK.
5. Select `*.common.jar` and *`.service.jar` files in the Package name or file name list.
6. Click Create and Validate Service Package.

   After creating and validating the service package, the following message is displayed:

   `Compressed` *`package_path`*`/SampleAppService.zip successfully!`

   `The service package is valid and it is ready to be deployed to DE.`
7. Click Finish. The next step is to create an application profile and register your application.

Add the application to Symphony DE

1. In the Package Explorer, right-click on project SampleApp. Select Symphony Operations > Add/ Remove Application.

   The Configure Symphony Project Page displays.
2. Click Validate to test the port connection to the Symphony DE PMC. (The plug-in "pings" the URL of the PMC to verify the connection.)

   Note:

   If the port is already in use, specify a different port number in file `vem_resource.conf` and enter it in the **Please specify the port number** textbox.
3. Once the PMC server replies, click OK.
4. In the Package Explorer, right-click on project SampleApp. Select Symphony Operations > Add/ Remove Application.

   The DE PMC displays.
5. Select Add an application. Click Continue.
6. Select Use existing profile and add application wizard. Click Browse and navigate to the location of the `SampleAppJava.xml` application profile. Click Continue.
7. Click Browse and navigate to the location of the service package. Select the service package file `SampleAppService.zip`. Click Continue.
8. Click Confirm to accept the application profile definitions.

   The project wizard creates your application and registers it within Symphony.
9. Click Close.
10. You can now run your client and submit workload to your Symphony application. In the Package Explorer, right-click com.platform.symphony.samples.SampleApp.client class. Select Run As >

Java Application. In the Select Java Application dialog, select SyncClient - com.platform.sysmphony.samples.SampleApp.client. Click OK.

Modifying existing applications

If you need to modify a service package or an application profile, refer to the Symphony DE PMC help for more information.

Visual Studio Tutorial

Tutorial: On-boarding a Symphony application with Visual Studio

Goal

This tutorial walks you through the steps for on-boarding a sample application using the Symphony add-in for Visual Studio. After completing the tutorial, your sample application will be able to perform calculations in parallel on the grid. Familiarity with the Visual Studio IDE is recommended.

Prerequisites

The prerequisites for on-boarding the sample application are:

- Visual Studio 2008 Professional/2010 Professional
- Symphony DE 5.1 or higher installed and running

Install the Symphony add-in for Visual Studio

The Symphony add-in and extensions are automatically installed in Visual Studio during installation of the Symphony DE package when the Visual Studio add-in option is selected. Visual Studio 2008 or 2010 must be installed on the development host prior to installing Symphony DE.

Sample applications

Two on-boarding application samples are provided with the Symphony DE package. One sample is a basic calculator program that calculates the interest for different principal amounts, interest rates, and durations. This sample is organized into a main program and a separate class library where the calculations are performed. In the context of Symphony, the main program and class library represent the client and service, respectively. This sample also serves as a foundation for this tutorial as you prepare it for on-boarding onto the grid.

The second sample reflects the same application as in the first sample, but it has already been updated and grid-enabled. This sample shows you what your first sample should look like once you have completed the on-boarding process. Use this sample to see how calculations are performed on the grid or as a handy reference if you encounter any difficulty on-boarding the first sample.

Note:

The code samples are intended to be run in a 32-bit environment only.

The sample applications are located at:

`%SOAM_HOME%\5.1\samples\AppOnboarding\DotNet\CS\BasicCalculator`.

About this tutorial

This tutorial refers to the sample applications included in the Symphony DE package at:

- `%SOAM_HOME%\5.1\samples\AppOnboarding\DotNet\CS\BasicCalculator\1-Before` (sample before on-boarding process)

- `%SOAM_HOME%\5.1\samples\AppOnboarding\DotNet\CS\BasicCalculator\2-After` (sample after on-boarding process, including optimization)

This tutorial was prepared using the Visual Studio 2008 Professional version but it should equally apply to Visual Studio 2010 Professional; any differences between the two Visual Studio versions relevant to this tutorial will be noted.

On-board an existing C#.NET application

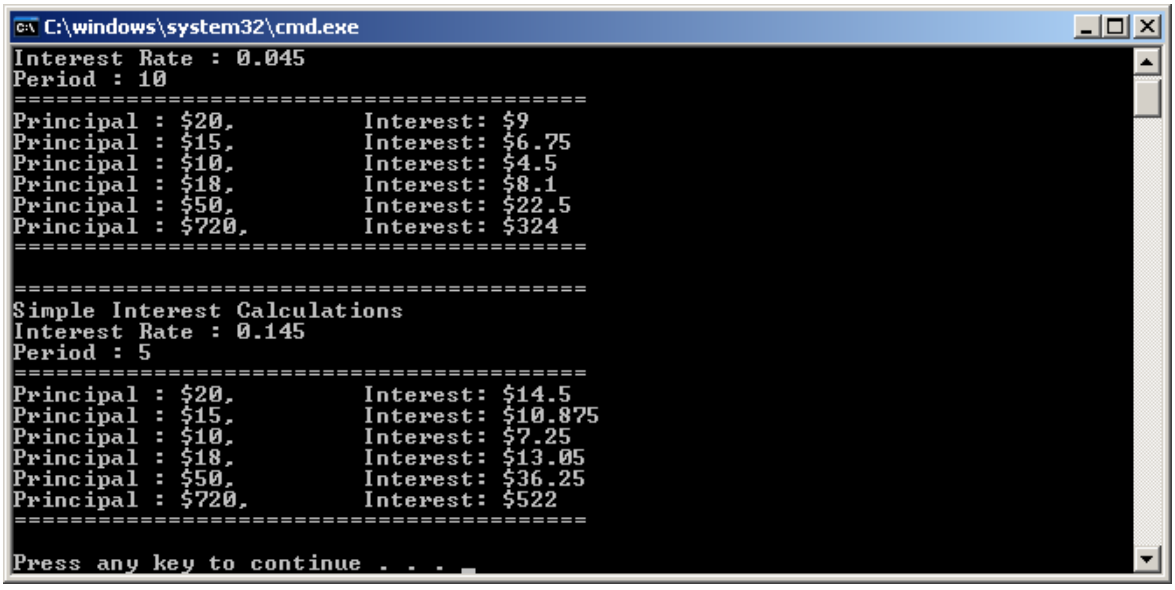
This section describes the steps for on-boarding an existing C#.NET application using the Symphony project wizard in Visual Studio.

Step 1: Test the sample

Before you start the on-boarding process, run the sample application to ensure your Visual Studio development environment is working properly.

1. Locate the sample solution folder at `%SOAM_HOME%\5.1\samples\AppOnboarding\DotNet\CS\BasicCalculator\1-Before`. If you are using Visual Studio 2008, double-click `BasicCalculator_2008.sln` to open the solution; if you are using Visual Studio 2010, double-click `BasicCalculator_2010.sln`.
2. Select Debug > Start Without Debugging.

   Visual Studio builds the solution and executes the code.
3. If your command window displays the following output, your development environment is working properly and you are all set to begin the on-boarding process. Proceed to Step 2.



Step 2: Add the .NET class library to the solution

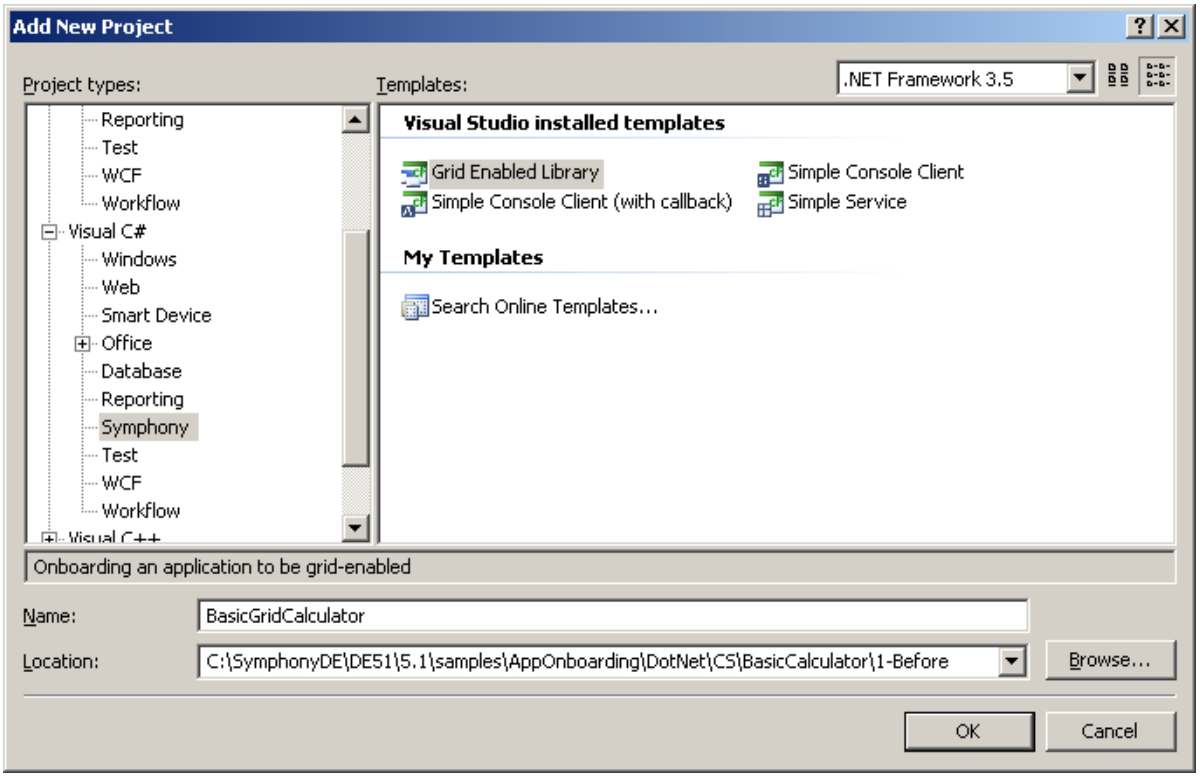
The following steps show you how to take the .NET class library that you built during the previous step and add it as a project to the sample solution.

Important:

It is recommended that you back up the sample Visual Studio solution before proceeding with the following steps in case you need to revert to the original files.

1. In the Solution Explorer, right-click the solution. Select Add > New Project.

   The Add New Project dialog displays.



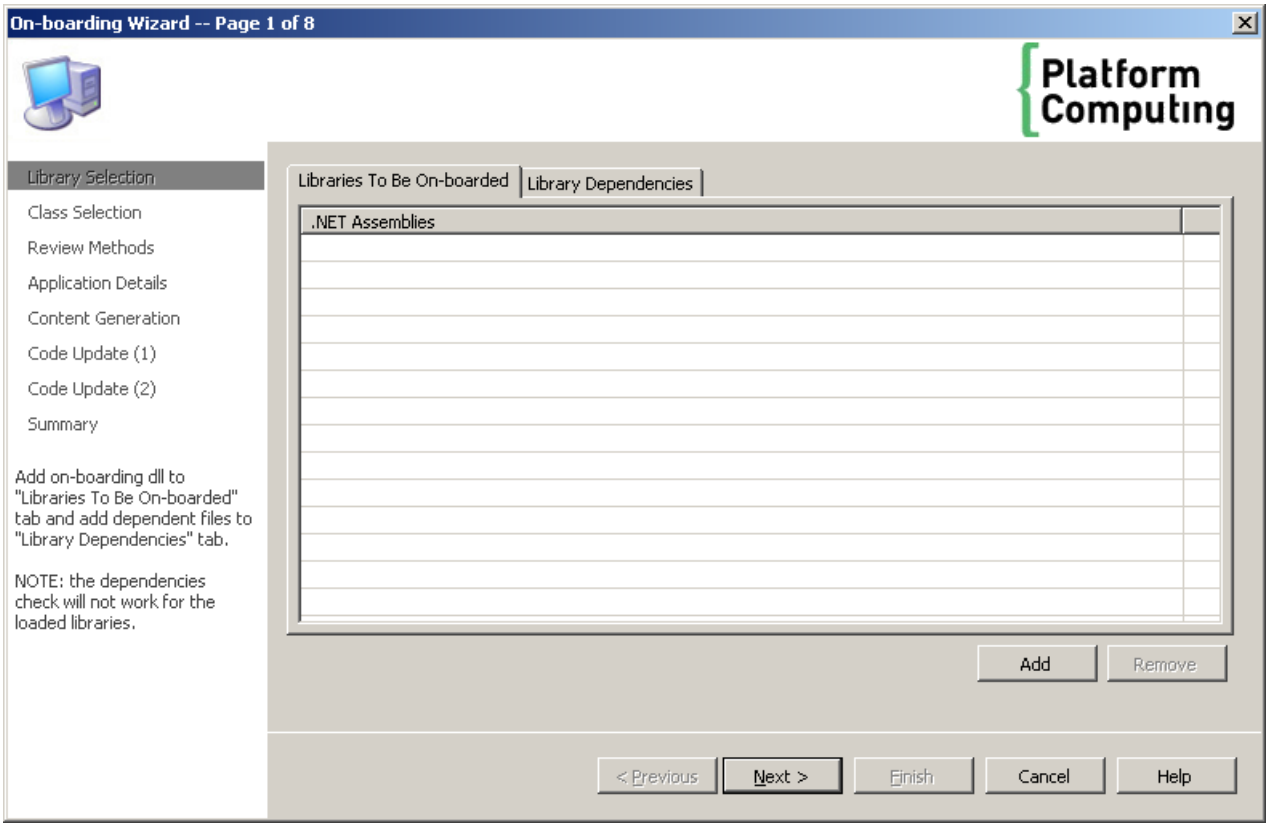
2. Make the following selections:
   a) In the Project types pane, select Symphony.
   b) In the Templates pane, select Grid Enabled Library.
3. In the Name textbox, enter **BasicGridCalculator**.

   Note:

   The project name is used as a namespace in the generated code. To avoid potential conflicts, do not choose a project name that is used as a namespace or class in the original code.

4. Click OK.

   The on-boarding wizard launches.



The wizard adds the following three projects to the solution:

- BasicGridCalculatorProxy (proxy class)
- BasicGridCalculatorService (computational logic)
- BasicGridCalculatorTransport (serializes/de-serializes proxy calls to/from the service)

5. Click Add.

   The Open file dialog displays.

6. Select the .NET class library `BasicCalculator.dll` that you built previously at `BasicCalculator\bin\Debug\`.
7. Click Open.

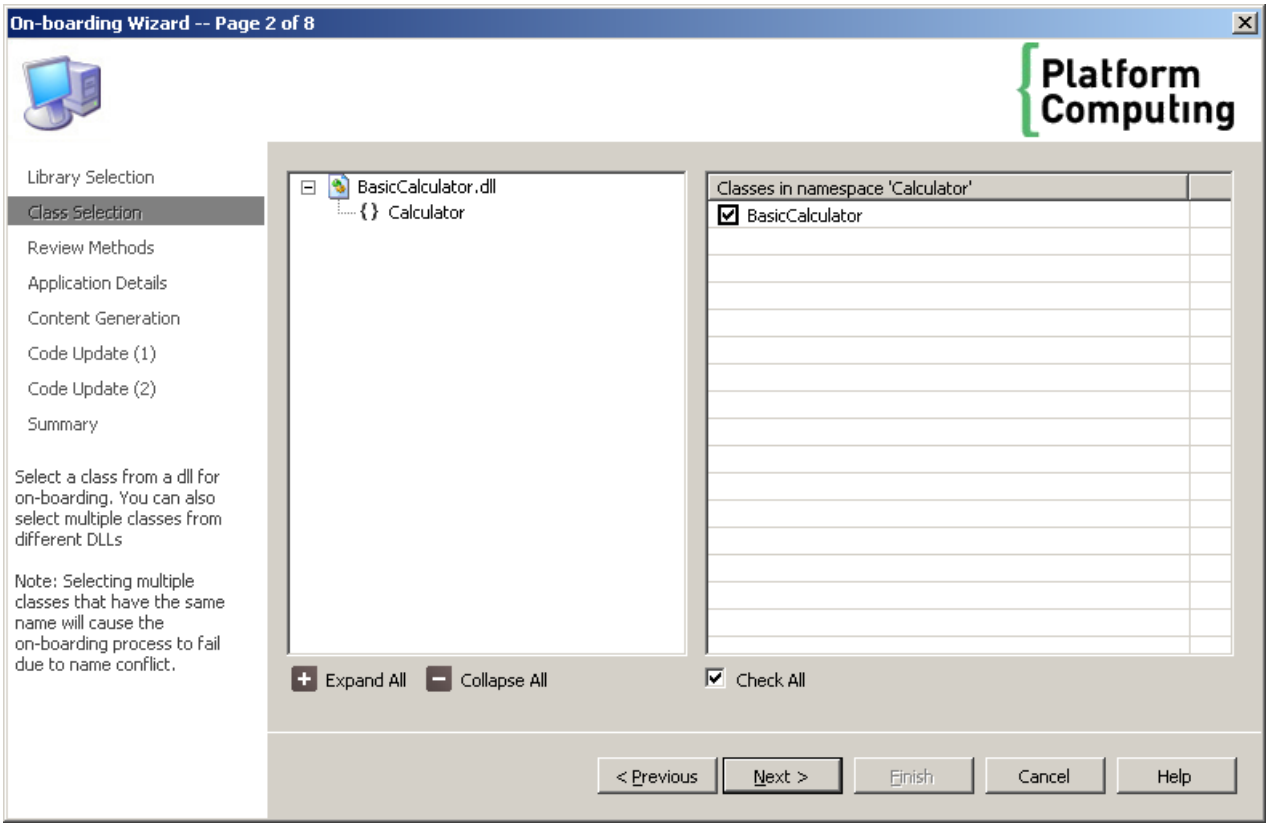
   The .NET class library is added to the list of .NET assemblies in the wizard.

8. If we had additional items such as dependent libraries or configuration files, we would select the Library Dependencies tab and add them here. Click Next. Proceed to Step 3.

Step 3: Expose the members in the .NET class library so that you can access them via the proxy

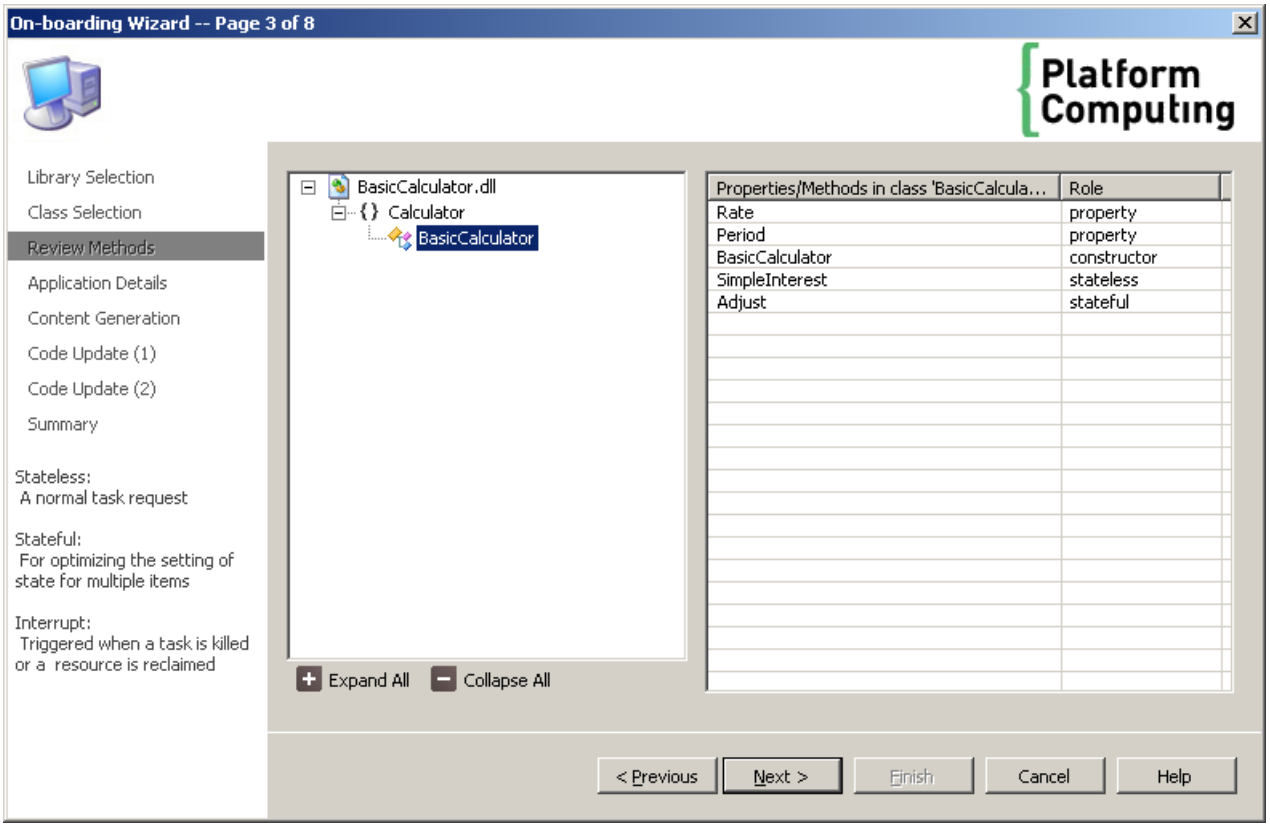
Exposing the methods, and properties of the .NET class library enables you to access them in the proxy so that you can interact with the grid-enabled class library as if it was a local object. Note that the wizard exposes all members of a selected class.

1. In the wizard's object explorer, expand `BasicCalculator.dll`. Select the Calculator namespace. Ensure BasicCalculator under Classes in namespace "Calculator" is checked.

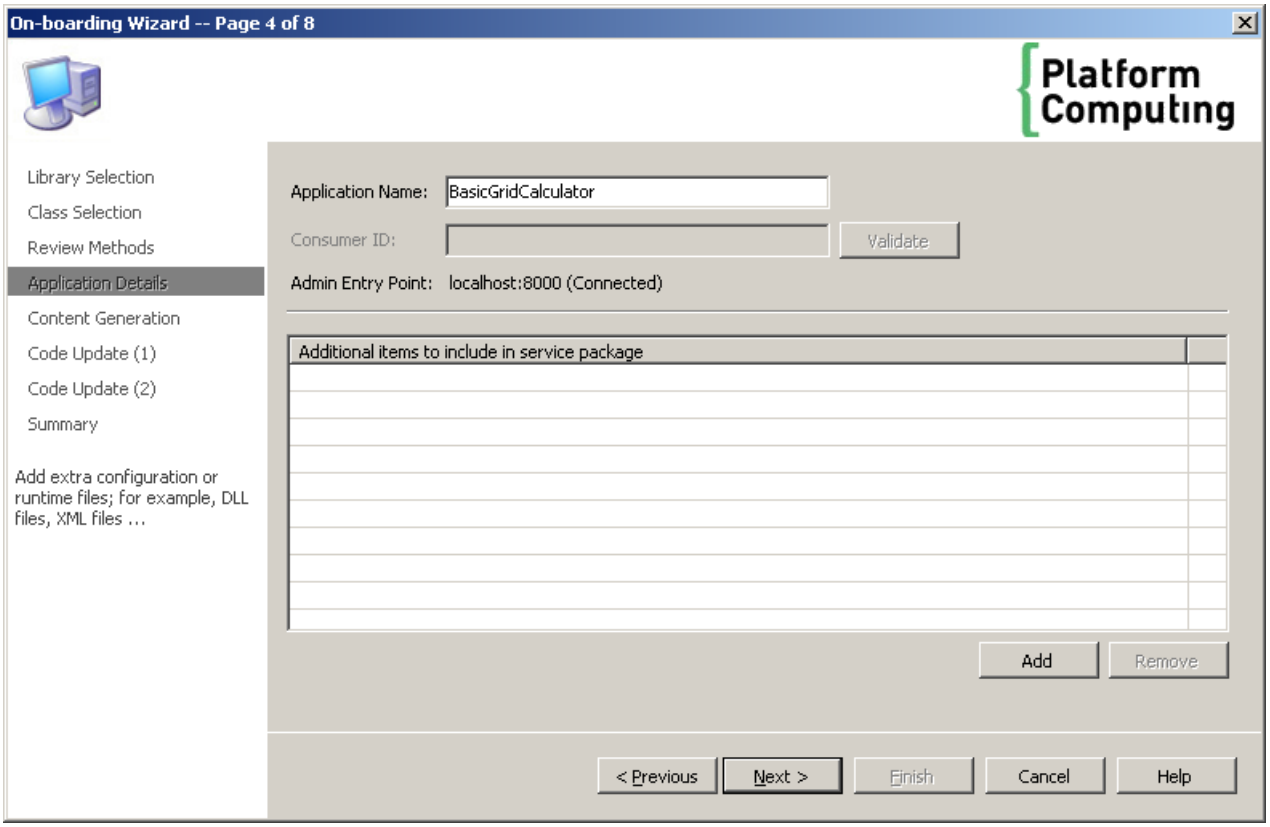


2. Click Next.
3. Fully expand BasicCalculator.dll and select BasicCalculator class.

   A list of the class's properties/methods and their respective roles displays. These are the methods and properties that will be exposed in the proxy object so that you can interact with the service; refer to the *Simplified application on-boarding with Visual Studio* feature reference in the *Application Development Guide* for a detailed description of property/method roles.



4. Click Next.
5. In the Application Name textbox, enter **BasicGridCalculatorApp**. If we were connected to the grid, we would also need to specify a valid consumer to which the application belongs.



6. Click Next.

   The wizard completes the code and project generation process and summarizes the generation activities. The sample application is registered and deployed to the grid.

7. Click Next.

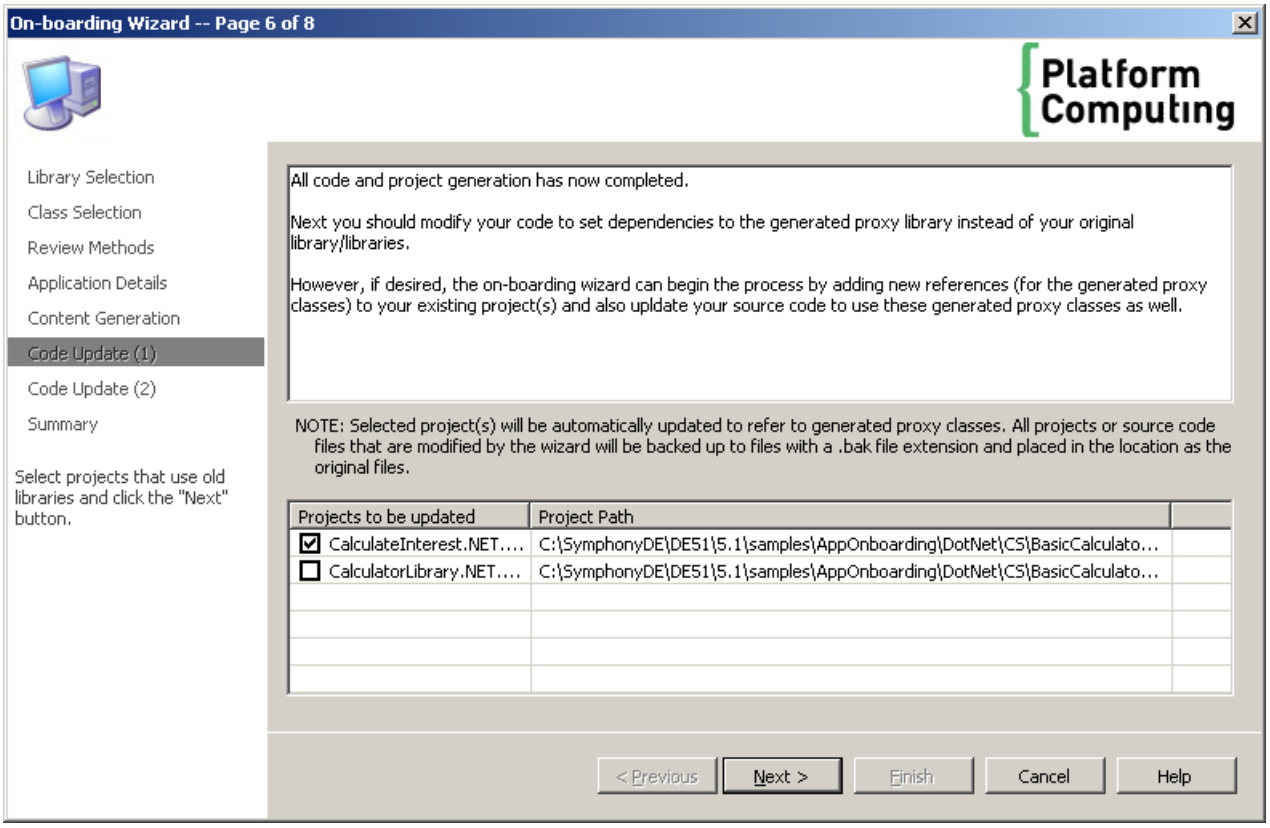
8. At this point, all code and project generation is complete but if you were able to run the client program `CalculateInterest.exe`, it would still call the methods of the local .NET class library instead of the library on the grid. You can either manually add the proxy reference to the client and update the code to refer to the generated proxy class or you can let the wizard do it for you. Let's have the wizard do the changes for us.

   In the Projects to be updated list, select CalculateInterest.NET.2008.

   Note:

   During this step, it is important to select only the projects where you want the wizard to update the client code and add references to the

proxy that was just generated and not select any of the other libraries currently being on-boarded.



9. Click Next.

   The wizard updates and builds the CalculateInterest.NET.2008 project.

   If you were to look at the code in the `Program.cs` file, you would see that a reference to the `BasicGridCalculator` class has been added and we are now using the BasicCalculator proxy class in our code instead..

```
using BasicGridCalculator;
...
//Main program
...
calculator = new BasicGridCalculator.BasicCalculator();
```

10. Click Next.

    We have now reached the last page of the wizard. All the activities performed by the wizard are recorded in `build.log` at the following locations:

    1. proxy build log

       `%SOAM_HOME%\5.1\samples\AppOnboarding\DotNet\CS\BasicCalculator\1-Before\BasicGridCalculator\BasicGridCalculatorProxy`

    2. service build log

       `%SOAM_HOME%\5.1\samples\AppOnboarding\DotNet\CS\BasicCalculator\1-Before\BasicGridCalculator\BasicGridCalculatorService`

    3. transport build log

```
%SOAM_HOME%\5.1\samples\AppOnboarding\DotNet\CS\BasicCalculator\1-
Before\BasicGridCalculator\BasicGridCalculatorTransport
```

11. Check that both Open the generated Readme.html after the wizard exits and Open the generated Report.txt after the wizard exits are selected.
12. Click Finish.

    The wizard closes and the `Readme.html` and `Report.txt` pages are displayed. The Readme file provides descriptions and code samples of the properties and methods that are available through the proxy. The Readme uses a color-coding scheme to distinguish the classes and methods, as follows:

    - Blue: links to proxy classes or methods that are derived from the original user-defined classes or methods
    - Green: links to classes or methods on these proxy objects that have been added by the wizard to make the classes or methods more grid-capable.

    The `Report.txt` file summarizes all the activities performed by the wizard during the on-boarding process including client code updates, creating and building projects, deploying service packages, and registering the application.

    Proceed to Step 4

Step 4: Add configuration files to handle mixed-mode assemblies

This step applies only to the Visual Studio 2010 IDE. If you are using Visual Studio 2008, proceed to Step 5.

As of .NET runtime 4.0, a change was made to no longer load mixed assemblies automatically. In order to load mixed assemblies, the runtime must be explicitly instructed through the use of a configuration file.Since all on-boarded applications still rely on the Symphony API (which is implemented as a mixed assembly), both the client and the service package to be deployed to the grid must be associated with a configuration file to instruct the runtime to load the mixed assembly.. This means that as long as a .Net binary is being loaded by the .NET 4.0 framework, as a minimum, the startup attribute useLegacyV2RuntimeActivationPolicy must be set to true in the configuration file. For convenience, we will use the configuration files that are packaged with the sample.

1. To associate a ".config" file with the client, copy the file `CalculateInterest.exe.config` located at `%SOAM_HOME%\5.1\samples\AppOnboarding\DotNet\CS\BasicCalculator\2-After\CalculateInterest` to the same location as the `Calculate.exe` executable.
2. To associate a ".config" file with the service that will run on the grid, add file `BasicGridCalculatorService.exe.config` to the service package on the grid.
   a) Right click on the BasicGridCalculatorService project.

      The project context menu displays.
   b) Select Platform Symphony > Application Details.

      The Application Details dialog displays.
   c) Under the section Files included in package, click the Add button.
   d) Browse and select the configuration file `BasicGridCalculatorService.exe.config` located at `%SOAM_HOME%\5.1\samples\AppOnboarding\dotnet\CS\BasicCalculator\2-After\BasicGridCalculator\BasicGridCalculatorService`.
   e) Click Create and Deploy Service Package.

      The service package is re-deployed to the grid.
3. Proceed to Step 5.

Step 5: Run the sample on the grid

Before we go any further, let's test the application on the grid to ensure it is working properly.

1. Select Debug > Start Without Debugging.

   Visual Studio builds the solution and executes the code.
2. Verify that your command window displays the same output as in Step 1.
3. You can also verify that the workload was successfully completed by accessing the Platform Management Console (PMC) through the Symphony menu extensions for Visual Studio. Before you can connect to the PMC, you must first select the respective service for the application. This is necessary since each service project stores its respective application profile and service package and an on-boarded application can generate multiple service projects.

   In the Solution Explorer, select the BasicGridCalculatorService project.
4. Select Symphony > Platform Management Console > Monitor Workload.
5. In the PMC, check that the Sessions page shows 1 session with 16 tasks done. For an explanation on how on-boarded application methods are mapped to workload, refer to *Simplified application on-boarding with Visual Studio* in the *Application Development Guide*.

   Proceed to Step 6.

Step 6: Free up Symphony resources

It is good programming practice to free up Symphony resources when they are no longer needed. For example, the client may have used a proxy and has no intention of using it anymore but the client is expected to run for several minutes after that. During this time, the session would remain open unnecessarily in Symphony. To address this issue, we can call the `Dispose()` method on the proxy object at a point where we no longer need the proxy and just before the reference goes out of scope.

```
...
     Calculate(principalAmounts, calculator);
}
catch (Exception E)
{
     Console.WriteLine("{0}\n{1}", E.Message, E.StackTrace);
}
finally
{
    if (null != calculator)
                {
                    calculator.Dispose();
                }
}
```

Refer to *Simplified application on-boarding with Visual Studio* in the *Application Development Guide* for more information about the `Dispose()` method.

Next, we will look at how to optimize the client code to maximize the benefits of running our application on the grid.

Step 7: Optimizing the code for the grid

In the previous step where we ran our application on the grid, our client interacted with the grid-enabled library and the calculations were actually performed on the grid. But the interaction between the client and the service was synchronous in nature as each input task was sent to the service, one at a time, in a blocking mode.

The following code snippet was taken from the client file `Program.cs`.

```
foreach (double amount in principalAmounts)
```

```
{
    double interest = calculator.SimpleInterest(amount);
    Console.WriteLine("Principal : ${0},\t Interest: ${1}",
    amount, interest);
}
```

In the code, we can see that the main execution thread loops through the principal amounts, one at a time, and waits for the result of each calculation to return before it calls the next method. Although the code executes and results are collected, this approach does not benefit from running on the grid. Benefits can only be realized when you execute methods in parallel across multiple computing resources available on the grid.

Asynchronous Programming Model

One of the simple ways to run our calculations in parallel is by using the .NET Asynchronous Programming Model (APM) convention and break our loop into two steps that use the appropriate methods to begin and end our calculation methods. In the code snippet from the previous section, instead of using the `SimpleInterest()` method on our proxy, we will use the `BeginSimpleInterest()` method that will not block our execution thread. This will allow us to submit multiple calculations, wait for each one to complete, and then collect the results from our `EndSimpleInterest()` method. Using the APM model, we can use a callback to collect the results as well.

Performing calculations using APM

Now let's review the same client code as before but this time we have updated it to use APM to perform the calculations in parallel. (Note that this code is taken from the Visual Studio solution at `%SOAM_HOME%\5.1\samples\AppOnboarding\DotNet\CS\BasicCalculator\2-After`.)

```
public static void CalculateInParallel(List<double>
principalAmounts, BasicGridCalculator.BasicCalculator calculator)
{
         ...
            // Begin calculations in parallel
            List<IAsyncResult> results = new
            List<IAsyncResult>(principalAmounts.Count);
            foreach (double amount in principalAmounts)
            {
                IAsyncResult result =
                calculator.BeginSimpleInterest(amount, null, amount
                as Object);
                results.Add(result);
            }
            foreach (IAsyncResult result in results)
            {
                result.AsyncWaitHandle.WaitOne();
                double interest = calculator.EndSimpleInterest
                (result);
                double amount = (double)result.AsyncState;
                Console.WriteLine("Principal : ${0},\t Interest:
                ${1}", amount, interest);
            }
}
```

To perform the calculations in parallel, first we need to set up a list of IAsyncResult objects to hold the results as they are collected asynchronously. The IAsyncResult objects store information about each interest calculation to be performed in the loop.. We will use this information to help us retrieve the results later in the second loop.

The `BeginSimpleInterest()` method takes three arguments: the principal amount, null (since we are not using any callback), and the principal amount typed as an object (we are using this as a convenient way to preserve the principal amount so we can retrieve it later when we print the output).

After calling `BeginSimpleInterest()`, the main thread continues execution while the asynchronous interest calculation takes place on the grid in the service process. Once executed, the

`BeginSimpleInterest()` immediately returns a reference that is added to the results list so that we can keep track of the corresponding result. The main thread cycles through the loop repeating the same sequence for each principal amount.

Once all the asynchronous interest calculations are done, a second loop retrieves the results. When we reach `result.AsyncWaitHandle.WaitOne()`, the main thread is blocked until the specific result is available. We retrieve the result by passing the result reference to the `EndSimpleInterest()` method. We also retrieve the original principal amount from the `AsyncState` property of the result object.

Performing calculations using APM with callback

In this sample, we review how to perform the calculations in parallel but this time we collect the results using a callback to let us know when the results are ready. The following sample code is taken from the Visual Studio solution at `%SOAM_HOME%\5.1\samples\AppOnboarding\DotNet\CS\BasicCalculator\2-After`.

```
public static void CalculateInParallelWithCallback(List<double>
principalAmounts, BasicGridCalculator.BasicCalculator calculator)
        {
            lock (resultLock)
            {
                outstandingMethodResults = principalAmounts.Count;
            }
            // Begin calculations in parallel (with a callback)
            foreach (double amount in principalAmounts)
            {
                SimpleInterestMethodContext ctx =
                new SimpleInterestMethodContext(calculator, amount);
                calculator.BeginSimpleInterest(amount,
                SimpleInterestCompleted, ctx as Object);
            }
            outstandingMethodsCompleted.WaitOne();
            outstandingMethodsCompleted.Reset();
        }
```

The first step is to initialize `outstandingMethodResults` to the number of expected results. We will use this variable to keep track of the number of results collected so that we know when all the work is done.

This time we construct a context object for each call in the loop since we will be processing the result in our callback method. As the callback will be called from a different thread, it is important to maintain a reference to our proxy so that we can still collect the results when the `SimpleInterest()` method completes later.

We pass three arguments to the `BeginSimpleInterest()` method: the principal amount, the delegate of the callback method, and the context object. When the call completes it will automatically call the callback that is supplied. At this point, the main thread waits until all the results are returned from the service.

Now let's take a look at the callback. With an asynchronous client, when a calculation is completed by the service, there must be a means of communicating this status back to the client. The callback (or response handler) is implemented for this purpose. It is called by the Middleware each time the service completes a calculation. In the following sample, `SimpleInterestCompleted()` is the callback method. It accepts the result object, which is passed to the method by the middleware whenever the respective calculation has completed.

```
private static void SimpleInterestCompleted(IAsyncResult result)
        {
            SimpleInterestMethodContext ctx =
            (SimpleInterestMethodContext)result.AsyncState;
            BasicGridCalculator.BasicCalculator calculator = ctx.obj;
            double amount = ctx.amount;
            try
```

```
            {
                double interest = calculator.EndSimpleInterest(result);
                Console.WriteLine("Principal : ${0},\t Interest: ${1}",
              amount, interest);
            }
            catch (Exception ex)
            {
                Console.WriteLine("Exception Occured in
                SimpleInterestCompleted Callback : {0}",
                ex.ToString());
            }
            finally
            {
                lock (resultLock)
                {
                    outstandingMethodResults--;
                    if (outstandingMethodResults < 1)
                    {
                        outstandingMethodsCompleted.Set();
                    }
                }
            }
        }
```

First, we restore the original context for the calculation. Next, we store a reference to the proxy on our result object so that we can make the call to `calculator.EndSimpleInterest()` when the callback is triggered from another thread. The result is retrieved by passing the result reference as a token to the `EndSimpleInterest()` method.

We count down the number of results outstanding. When all the results have returned, we set a flag, which unblocks the main thread in the `CalculateInParallelWithCallback()` method.

Step 8: Run the optimized sample on the grid

For instructions on how to run the optimized grid sample, refer to the Basic Calculator (After on-boarding) Readme located at `%SOAM_HOME%\5.1\samples\AppOnboarding\DotNet\CS\BasicCalculator\2-After`.

Index

A

asynchronous client
  tutorial 13
  tutorial, Java 45, 77

C

client
    samples
      asynchronous client 13
      asynchronous client for linux 14, 45, 54
      asynchronous client for Windows 13
      Java, asynchronous client 45, 77
      Java, synchronous client for Windows 34
      synchronous client for linux 5, 22, 34, 92
      synchronous client for Windows 5, 22, 91
clients
    samples
      sharing data 28
code samples
  asynchronous client 13
    Java
      sharing data, client and service 59, 83, 91
  Java, asynchronous client 77
  sharing data, client and service 28
  synchronous client 8, 16, 25, 31, 37, 48, 56, 62, 97–99
code samples, Java
  asynchronous client 45

D

data
  Java tutorial for sharing data among tasks 59, 83, 91
  tutorial for sharing data among tasks 28

M

message object
    tutorial
      declare 8

S

samples
  asynchronous client 13
    Java
      service 53
  Java, asynchronous client 45, 77
  service 22
  sharing data, client and service 28
service
    samples
      asynchronous client for linux 14, 45, 54
      asynchronous client for Windows 13
      deploy for data sharing 29
      Java, synchronous client for Windows 34
      synchronous client for linux 5, 22, 34, 92
      synchronous client for Windows 5, 22, 91
services
    samples
      sharing data 28
session description
  in client code 11, 21, 43, 121
session type
  in client code 11, 21, 43, 122

T

tutorial
  asynchronous client 13
  basic service 5, 22
    Java
      basic service 53
      sharing data among tasks 59, 83, 91
      synchronous client 34
  Java, asynchronous client 45, 77
  sharing data among tasks 28
  synchronous client 5