Communicating between Vision & Oracle using API

Procedure Document

TABLE OF CONTENTS

1 Communicating between Vision & Oracle using API

1.1 Overview

Beginning in 2007, ABC Company purchased Oracle to store HRMS and accounting transactions. Additionally, ABC Company purchased Vision, a SQL Server database application, as a repository for marketing and project management transactions.

With the migration of business data to Vision and Oracle, IT needed a means to integrate with and between two disparate database engines— SQL Server (for Vision) and Oracle. In response, Applications Development Group provided a set of Shared Application Components that function as a bridge between Vision and Oracle.

Important note:

This *fundamental* Shared Application Components project opens the way for ABC Company to bridge databases and applications other than Vision and Oracle.

1.1.1 Target Readers and Users

This document is provided for Applications Development Group developers and other IT personnel who need to transfer data between SQL Server and Oracle.

1.1.2 Definitions, Acronyms and Abbreviations

The following terms and acronyms apply to this project.

Term	Definition
ADG	Applications Development Group
API	Application Program Interface. This document refers to “Shared Application Components” as API.

1.1.3 Prerequisites

- Microsoft Windows 2000 operating system as the platform for all development, staging, and production servers across the ABC Company network.
- Windows Services FileDispatcher and TimeDispatcher.
- Visual Studio 2005 libraries containing Vision and Oracle Business Objects
- Two folders in known network locations

  A named folder that holds data waiting for retrieval

  An archive folder where retrieved data moves after transmission
- Stored Procedures within SQL Server and Oracle accept and process new or changed data.

2 Procedure

2.1 Overview

Windows Services include

- FileDispatcher, which transfers data on-demand. Typically, on-demand needs occur when an application cannot proceed until it receives external data.

  For example, when management approves the change in status of an Opportunity to a Project, Vision makes an on-demand request to Oracle for a project number.
- TimeDispatcher, which synchronizes data between Oracle and Vision at scheduled intervals.

  For example, Oracle retains employee data. Any new project-management employee must have a Vision account. At the interval specified, TimeDispatcher triggers a synchronizer object that can convert new Oracle employee objects to Vision-compatible form.

This document describes how a developer:

- Configures Service X to trigger communications between the database engines
- Provides entry points to objects in the application that consume the data communicated

2.1.1 VSS Structure for API

- ABC.Services holds the configuration and object code for Windows Services, including FileDispatcher and TimeDispatcher.

  Start at **ABC.Services\Windows.** Under Windows, you might have to drill down through several nested, empty folders to reach **FileDispatcher.config** and **TimeDispatcher.config** files you need to edit for your application to work.

ABC.Services examples in this document are located in
\ABC.Services\Windows\Dispatcher Services\Dispatcher Service\Dispatcher Deamon-Service\Dispatcher Service\Dispatcher Service

- ABC.Controllers holds the business objects and synchronizer objects for Vision and Oracle.

  FileDispatcher runs the ABC.Controller objects you specify in the related FileDispatcher.config;

  TimeDispatcher runs the ABC.Controller and ABC.Controller.Synchronize objects you specify in TimeDispatcher.config.

2.2 .Using FileDispatcher

2.2.1 Configuration

Sample 1: FileDispatcher.config

```
<?xml version="1.0" encoding="utf-8" ?>
<CONFIG>
  <ENVIRONMENT name ="STAGING">
    <FileWatchers ThreadSleepInterval="15">
      <Watcher HeartBeatInterval="15" ProcessTimeOut="1800" Path
="c:\inetpub\ftproot\localuser\filedropuser\" IncomingPattern="\bvision_"
ArchivingFolder="c:\inetpub\ftproot\localuser\filedropuser\archive\"
TaskTypeToRun="ABC.Controller.VisionProjectController" TaskAssembly="ABC.Controller.Vision.Project"
FailureTaskTypeToRun="" FailureTaskAssembly=""/>
    </FileWatchers>
  </ENVIRONMENT>
  <ENVIRONMENT name ="DEVELOPMENT">
    <FileWatchers ThreadSleepInterval="15">
      <Watcher HeartBeatInterval="15" ProcessTimeOut="1800" Path ="c:\VisionMsg\"
IncomingPattern="\bvision_" ArchivingFolder="c:\VisionMsg\Archive\"
TaskTypeToRun="ABC.Controller.VisionProjectController" TaskAssembly="ABC.Controller.Vision.Project"
FailureTaskTypeToRun="" FailureTaskAssembly=""/>
    </FileWatchers>
  </ENVIRONMENT>
  <ENVIRONMENT name ="PROD">
    <FileWatchers ThreadSleepInterval="15">
      <Watcher HeartBeatInterval="15" ProcessTimeOut="1800" Path
="c:\inetpub\ftproot\localuser\filedropuser\" IncomingPattern="\bvision_"
ArchivingFolder="c:\inetpub\ftproot\localuser\filedropuser\archive\"
TaskTypeToRun="ABC.Controller.VisionProjectController" TaskAssembly="ABC.Controller.Vision.Project"
FailureTaskTypeToRun="" FailureTaskAssembly=""/>
    </FileWatchers>
  </ENVIRONMENT>
</CONFIG>
```

FileDispatcher.config specifies where to look in Development, Staging, and Production environments for:

- The shared directory and filename prefix containing on-demand data
- The archive directory that keeps historic data and enables retransmission if an error occurs
- Names of controller objects that consume the data.

As shown in Sample 1, only the Path to the shared and archive directories change with deployment to another environment.

Sample 2, below, details setup for Staging. You apply the same structure Development and Production, and change the Path.

Sample 2: Staging environment details

```
<ENVIRONMENT name ="STAGING">
 <FileWatchers ThreadSleepInterval="15"> //Suspends the current thread for a specified milliseconds
  <Watcher HeartBeatInterval="15" //How often, in milliseconds, the agent checks for new data
   ProcessTimeOut="1800" //Seconds before thread closes if no new data found
   Path ="c:\inetpub\ftproot\localuser\filedropuser\" //Shared folder holding new on-demand data
   IncomingPattern="\bvision_" //In Path specified above, look for files starting “bvision_”
   ArchivingFolder="c:\inetpub\ftproot\localuser\filedropuser\archive\" // move “bvision_” data here after send
   TaskTypeToRun="ABC.Controller.VisionProjectController" // TaskType that processes “bvison_” data
    TaskAssembly="ABC.Controller.Vision.Project"  // Directory where TaskTypeToRun is located
    FailureTaskTypeToRun="" FailureTaskAssembly=""/> //Error handling
 </FileWatchers>
</ENVIRONMENT>
```

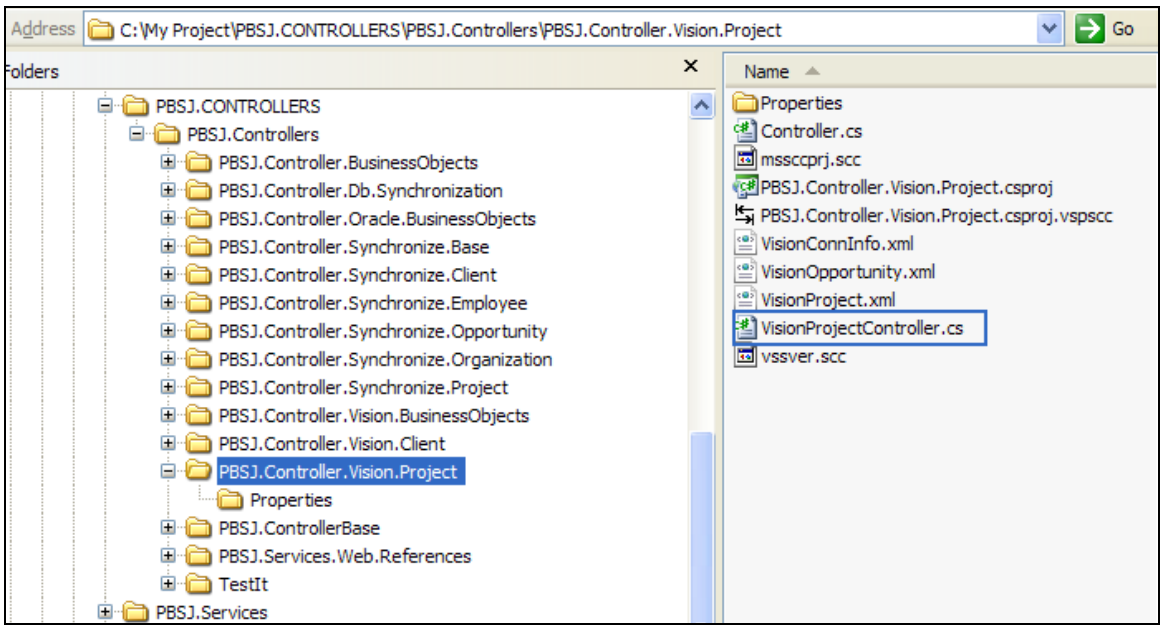
2.2.2 TaskTypeToRun in TaskAssembly

To take advantage of the FileDispatcher API, you must supply the following in your code:

- **PROCESS_CODE**, a constant value that uniquely identifies your object
- **IProcess** parameter that passes PROCESS_CODE.
- **Get** method that identifies your object’s PROCESS_CODE
- **Run** method that translates your object from Oracle to Vision, or from Vision to Oracle.

Our example: Assume the FileDispatcher finds a new file in shared folder **filedropuser**. FileDispatcher looks in **TaskAssembly** directory **ABC.Controll.Vision.Project** for an object called **VisionProjectController.** It finds and executes a C# object, **VisionProjectController.cs**

Sample 3: Finding ABC.Controller.Vision.Project\VisionProjectController



VisionProjectController.cs is a child of **ControllerBase.cs**. VisionProjectController.cs declares a unique value for constant *Process_Code* which distinguishes it from other Controller objects that inherit common properties from ControllerBase.

Sample 4: Passing IProcess to ControllerBase

```
namespace ABC.Controller
{ // IProcess determines which attributes VisionProjectController inherits or overrides from ControllerBase
   public class VisionProjectController : ControllerBase, IProcess
   {
      #region Constants
      const string PROCESS_CODE = "CONTR_PROJECT";
      public const String PARAM_FILE_NAME = "FileName";
      public const string PROJECT_TEMPLATE = "Billable Template";
      public const string PROPOSAL_TEMPLATE = "Proposal Template";
      private const string STAGE_CONTRACT = "PI001";
      private const string STAGE_PURSUIT = "Purs001";
      #endregion
 // Variables not shown
#region Constructors
      public VisionProjectController()
      {
         this.processCode = PROCESS_CODE;
      }
       #endregion
```

Sample 5 shows Process_Code passing between child and parent as parameter *IProcess*. Refer to the **get** command in ControllerBase method ProcessCode.

Sample 5: Parent ControllerBase applies Process_Code

```
namespace ABC.Controller
{
   public abstract class ControllerBase : IProcess
   {
  // more #region blocks that are not shown for this example
  #region IProcess Members
      public void Run(ProcessSession processSession, ABC.Common.BusinessObject.BoBase FamilyBO)
      { //Run is overridden in child VisionProjectController
         throw new Exception("The method or operation is not implemented.");
      }
      public string ProcessCode //get the ProcessCode from child object
      {
         get
         {
            return processCode;
         }
      }
      #endregion
   }
```

```
}
```

For this example, ignore the Run command in parent ControllerBase. VisionProjectController overrides the Run command as shown in Sample 6.

Sample 6: VisionProjectController Run method for CONTR_PROJECT

```
 namespace ABC.Controller
{
  public class VisionProjectController : ControllerBase, IProcess
  {
// scroll down to #region with Overrides at end of VisionProjectController.cs
//
//
    #region Overridden methods
    public new void Run(ProcessSession processSession, ABC.Common.BusinessObject.BoBase BO)
    {
      this.processSession = processSession;
      foreach (SessionParameter sessionParameter in this.processSession.GetParameters())
      {
        SetParameter(sessionParameter);
      }
      this.ProcessFile(this.PR_FileName);
    }
    private void SetParameter(SessionParameter parameter)
    {
      if (parameter.PR_Name == PARAM_FILE_NAME)
        this.PR_FileName = parameter.PR_Value;
    }
    #endregion
   }
}
```

2.3 Using TimeDispatcher

2.3.1 Configuration

Sample 7: TimeDispatcher.confi

```
<?xml version="1.0" encoding="utf-8" ?>

<CONFIG>

  <ENVIRONMENT name ="DEVELOPMENT">

    <TimeWatchers ThreadSleepInterval="15">

  <Watcher HeartBeatInterval="1000" DispatchInterval="60" IntervalType="SECOND" TaskTimeOut="1800"
TaskTypeToRun="ABC.Controller.Synchronize.Employee.Synchronizer"
TaskAssembly="ABC.Controller.Synchronize.Employee" />

 </TimeWatchers>

  </ENVIRONMENT>

  <ENVIRONMENT name ="STAGING">

   <TimeWatchers ThreadSleepInterval="15">

    <Watcher HeartBeatInterval="1000" DispatchInterval="600" IntervalType="SECOND"
TaskTimeOut="1800" TaskTypeToRun="ABC.Controller.Synchronize.Project.Synchronizer"
TaskAssembly="ABC.Controller.Synchronize.Project" />

    <Watcher HeartBeatInterval="1000" DispatchInterval="480" IntervalType="SECOND"
TaskTimeOut="1800" TaskTypeToRun="ABC.Controller.Synchronize.Client.Synchronizer"
TaskAssembly="ABC.Controller.Synchronize.Client" />

    <Watcher HeartBeatInterval="1000" DispatchInterval="300" IntervalType="SECOND"
TaskTimeOut="1800" TaskTypeToRun="ABC.Controller.Synchronize.Employee.Synchronizer"
TaskAssembly="ABC.Controller.Synchronize.Employee" />

    <Watcher HeartBeatInterval="1000" DispatchInterval="300" IntervalType="SECOND"
TaskTimeOut="1800" TaskTypeToRun="ABC.Controller.Synchronize.Employee.LanguageSynchronizer"
TaskAssembly="ABC.Controller.Synchronize.Employee" />

    <Watcher HeartBeatInterval="1000" DispatchInterval="300" IntervalType="SECOND"
TaskTimeOut="1800"
TaskTypeToRun="ABC.Controller.Synchronize.Employee.HrmsEducationSynchronizer"
TaskAssembly="ABC.Controller.Synchronize.Employee" />

    <Watcher HeartBeatInterval="1000" DispatchInterval="300" IntervalType="SECOND"
TaskTimeOut="1800"
TaskTypeToRun="ABC.Controller.Synchronize.Employee.HrmsRegistrationSynchronizer"
TaskAssembly="ABC.Controller.Synchronize.Employee" />

   </TimeWatchers>

  </ENVIRONMENT>
```

```
 <ENVIRONMENT name ="PROD">

  <Watcher HeartBeatInterval="1000" DispatchInterval="86400" IntervalType="SECOND"
TaskTimeOut="1800" TaskTypeToRun="ABC.Controller.Synchronize.Project.Synchronizer"
TaskAssembly="ABC.Controller.Synchronize.Project" />

  <Watcher HeartBeatInterval="1000" DispatchInterval="900" IntervalType="SECOND"
TaskTimeOut="1800" TaskTypeToRun="ABC.Controller.Synchronize.Client.Synchronizer"
TaskAssembly="ABC.Controller.Synchronize.Client" />

  <Watcher HeartBeatInterval="1000" DispatchInterval="86400" IntervalType="SECOND"
TaskTimeOut="1800" TaskTypeToRun="ABC.Controller.Synchronize.Employee.Synchronizer"
TaskAssembly="ABC.Controller.Synchronize.Employee" />

  <Watcher HeartBeatInterval="1000" DispatchInterval="86400" IntervalType="SECOND"
TaskTimeOut="1800" TaskTypeToRun="ABC.Controller.Synchronize.Employee.LanguageSynchronizer"
TaskAssembly="ABC.Controller.Synchronize.Employee" />

  <Watcher HeartBeatInterval="1000" DispatchInterval="86400" IntervalType="SECOND"
TaskTimeOut="1800"
TaskTypeToRun="ABC.Controller.Synchronize.Employee.HrmsEducationSynchronizer"
TaskAssembly="ABC.Controller.Synchronize.Employee" />

  <Watcher HeartBeatInterval="1000" DispatchInterval="86400" IntervalType="SECOND"
TaskTimeOut="1800"
TaskTypeToRun="ABC.Controller.Synchronize.Employee.HrmsRegistrationSynchronizer"
TaskAssembly="ABC.Controller.Synchronize.Employee" />

 </ENVIRONMENT>
</CONFIG>
```

TimeDispatcher.config specifies how TimeDispatcher should work in Development, Staging, and Production. In Sample 7, only the Employee synchronizer is still in development.

Staging specifies varying HeartBeatIntervals at which to check for changes in Projects, Clients, and Employees. Additionally, TimeDispatcher checks for updates to employee language skills and education because proposals list project team abilities. Project changes are updated in staging every 10 minutes (600 seconds), Client changes synchronize every 8 minutes (480 seconds), and employee data synchronizes every 5 minutes (300 seconds). In this way, the user can see the changes occur in the User Acceptance/Test environment on the Staging platform.

Production synchronizes the same objects as Staging, but the DispatchInterval lengthens to 24 hours (86400 seconds) in live production.

Sample 8 explains each property for a typical synchronizer, Project.Synchronizer.

Sample 8: Synchronize Project (Staging)

```
<ENVIRONMENT name ="STAGING">

 <TimeWatchers ThreadSleepInterval="15">//Suspends the current thread for a specified milliseconds

  <Watcher HeartBeatInterval="1000"  //Milliseconds between checks for new source data

     DispatchInterval="600" // How often to send new or changed data to recipient database
      IntervalType="SECOND"// DispatchInterval units

     TaskTimeOut="1800" //Seconds until thread closed if no new data found

     TaskTypeToRun="ABC.Controller.Synchronize.Project.Synchronizer"  //Task object that synchronizes
     TaskAssembly="ABC.Controller.Synchronize.Project" />// Directory where TaskTypeToRun is located
```

Sample 9: Synchronize Project (Production)

```
 <ENVIRONMENT name ="PROD">

  <TimeWatchers ThreadSleepInterval="15">//Suspends the current thread for a specified milliseconds

   <Watcher HeartBeatInterval="1000"  //Collect data every 1000 milliseconds

DispatchInterval="86400" //  ((86400 / 60) / 60) = 24 hours. Send data collected data to target once per day
 IntervalType="SECOND"// DispatchInterval units

TaskTimeOut="1800" //Seconds until thread closed if no new data found

TaskTypeToRun="ABC.Controller.Synchronize.Project.Synchronizer"  //Task object that synchronizes
TaskAssembly="ABC.Controller.Synchronize.Project" />// Directory where TaskTypeToRun is located
```

Sample 8 and Sample 9, above, Project's synchronizer collects source data every second (HeartBeatInterval of 1000 milliseconds).

- In the Staging environment, TimeDispatcher sends the collected data to the target database every 10 minutes.
- By contrast, in the Production environment, TimeDisplatcher waits 24 hours before sending the source data collected all day to the target database. Production dispatch occurs at night.

Example:

In this example, Oracle is the source database and Vision is the target database. The Project Manager assigned to Oracle project number 999999 changes. Project.Synchronizer translates the altered Oracle Business Object to a Vision Business Object and holds updated Vision-formatted project 999999 with other Vision projects collected since last night's Dispatch. At tonight's DispatchInterval, Vision receives the changed Project Manager for project 999999 and updates its tables accordingly.

2.3.2 TaskTypeToRun in TaskAssembly

To take advantage of the TimeDispatcher API, you supply the following in your code:

- **PROCESS_CODE**, a constant value that uniquely identifies your object
- **TaskAssemblies –** directories containing C# (.cs) or Visual Basic (.vb) **TasktoRun** objects.

  Each TasktoRun is part of an object hierarchy. The hierarchical family of objects provides all the structure and behavior needed for the application to interface and maintain integrity between disparate objects, such as Oracle and Vision databases.

The tasks run to maintain corresponding data in Vision and Oracle all use Controllers that Synchronize the data between these two databases. Each synchronizing controller task does the necessary conversion between Oracle and Vision, according to the differences between them in database schema.

This document describes one example of a TasktoRun—Synchronizer.cs in TaskAssembly ABC.Controller.SynchronizeProject.

Sample 10: Excerpt from ABC.Controller.Sychronize.Project.Synchronizer.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using OracleBO = ABC.Oracle.BusinessObjects;
using VisionBO = ABC.Vision.BusinessObjects;
using ABC.Common.BusinessObject; //Look in ABC.Common.Framework for any ABC.Common object.
                                 //ABC.Common.Framework contains TaskAsssembly ABC.Common.BusinessObject
                                 //Look here for definition of BoBase.cs (parent Business Object)
using ABC.Controller;

namespace ABC.Controller.Synchronize
{
  namespace Project
  {
    public class Synchronizer : SynchronizerBase //Synchronizer is a child of
                      // public abstract class SynchronizerBase : ControllerBase, IProcess
                      // SynchronizerBase is a child of ControllerBase, which needs IProcess
    {
      #region Constants
      const string PROCESS_CODE = "SYNC_PROJECT";
                        //Process_Code signifies action SYNC(hronize) to object PROJECT.
      #endregion

      #region Constructors
       public Synchronizer()
       {
        this.processCode = PROCESS_CODE;
       }
      #endregion

      protected override BusinessObjectCollection<BoBase> GetSourceCollection()
      {//GetSourceCollection() returns a collection of type BoBase objects,
       //all structured with ABC.Common.BusinessObject.BoBase.cs
        BusinessObjectCollection<OracleBO.Project> projectColl =
                                            OracleBO.Project.GetUpdatedProjects(PR_ObjectKey, PR_SecondsInterval);
        // Collect updates of type Oracle BusinessObject Project
        return new BusinessObjectCollection<BoBase>(projectColl.Ds.Tables[OracleBO.Project.TABLE_NAME]);
        // Append as Oracle BusinessObject Project to BusinessObjectCollection
      }
```

```
protected override BoBase GetTargetObject(BoBase sourceBo)
{
  try
  {//if collection found any updated Oracle BusinessObject Project
    OracleBO.Project oracleProject = new OracleBO.Project(sourceBo.Row);
 //grab the row as Oracle and return a Vision Business Object Project with the same project number.
    return new VisionBO.Project(oracleProject.PR_ProjectNumber);
  }
  catch (BODataNotFoundException)
  {
    return null;
  }
}
```

The remainder of Sychronizer.cs maps source fields to target.

IProcess, Get, and Run

Grandparent object ControllerBase gets Process_Code, in this case SYNC_PROJECT.

Parent object SychronizerBase Runs the Process_Code. The Run command in SynchronizerBase.cs instantiates a session. Project Synchronizer.cs functions inside of the SychronizerBase session.

Generic use of TimeDispatcher Windows Service

If your project uses TimeDispatcher for something other than Vision or Oracle, or both, your TasktoRun will have different logic. You will still need to declare a Process_Code constant, a get routine to pass along your Process_Code, and a Run routine to act based on the Process_Code value. get and run might occur anywhere along your hierarchy.

If you base your object on SynchronizeBase.cs, you might override Run in your object to do something other than establish a session.