Microsoft Dynamics® AX 2012

Introduction to the SysOperation framework

White Paper

This paper provides a comparison of the new SysOperation framework introduced in Microsoft Dynamics AX 2012 with the RunBase framework, and it outlines common development scenarios using the new framework.

Date: March 2013

www.microsoft.com/dynamics/ax

Arif Kureshy

Send suggestions and comments about this document to adocs@microsoft.com. Please include the title with your feedback.

Table of Contents

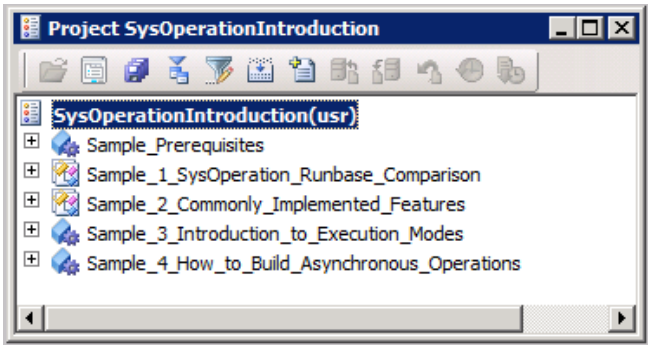
Introduction

SysOperation is a framework in Microsoft Dynamics® AX 2012 that allows application logic to be written in a way that supports running operations interactively or via the Microsoft Dynamics AX batch server. The framework provides capabilities that are very similar to the RunBase framework that came before it. The batch framework has very specific requirements for defining operations:

- The operation must support parameter serialization so that its parameters can be saved to the batch table.
- The operation must have a way to display a user interface that can be launched from the batch job configuration user interface.
- The operation must implement the interfaces needed for integration with the batch server runtime.

The RunBase framework defines coding patterns that implement these requirements. The SysOperation framework provides base implementations for many of the patterns defined by the RunBase framework.

The purpose of this white paper is to outline how the SysOperation framework can be used to build operations that can run asynchronously and make use of the full processing power available on the server. Five samples are presented: the first two demonstrate the basic concepts, the second two demonstrate how to build asynchronous scalable operations and the fifth sample shows how to call asynchronous operations from .NET clients. The following figure shows the X++ project containing the sample code for the first four samples. The fifth sample is a Microsoft® Visual Studio® solution.

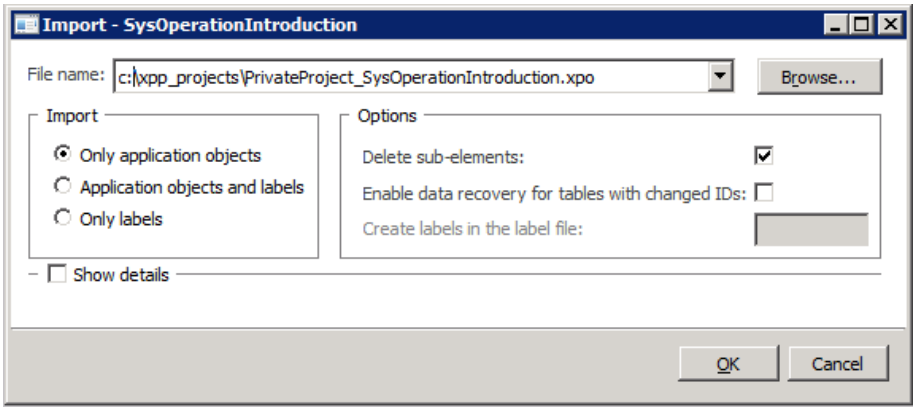


- **Sample 1** – Comparison of the SysOperation and RunBase frameworks
- **Sample 2** – Demonstration of commonly implemented features in SysOperation and RunBase
- **Sample 3** – Introduction to SysOperation execution modes
- **Sample 4** – How to build asynchronous operations with the SysOperation framework
- **Sample 5** – How to call asynchronous operations from .NET clients

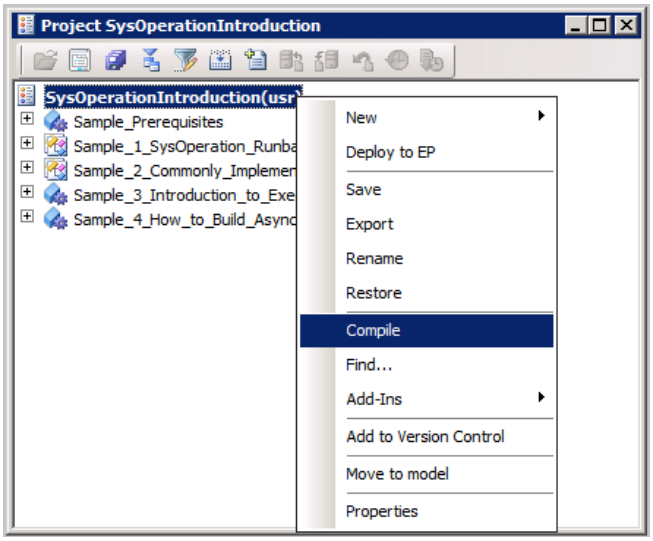
Loading the sample code

It is useful to run the samples in addition to reading this paper. To load and run the samples, unpack the X++ project associated with this paper, and then follow these steps:

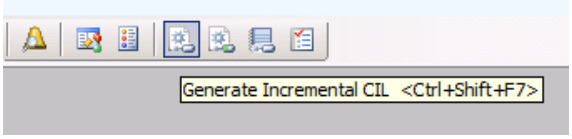
1. Download the zip file **Introduction to the SysOperation Framework.zip** from http://blogs.msdn.com/b/aif/archive/2012/03/17/introduction-to-the-sysoperation-framework.aspx, expand it and then copy the contents to the machine where the Microsoft Dynamics AX is installed.
2. Import the project file, PrivateProject_SysOperationIntroduction.xpo. It does not overlay any existing framework classes, so there should be no conflicts.



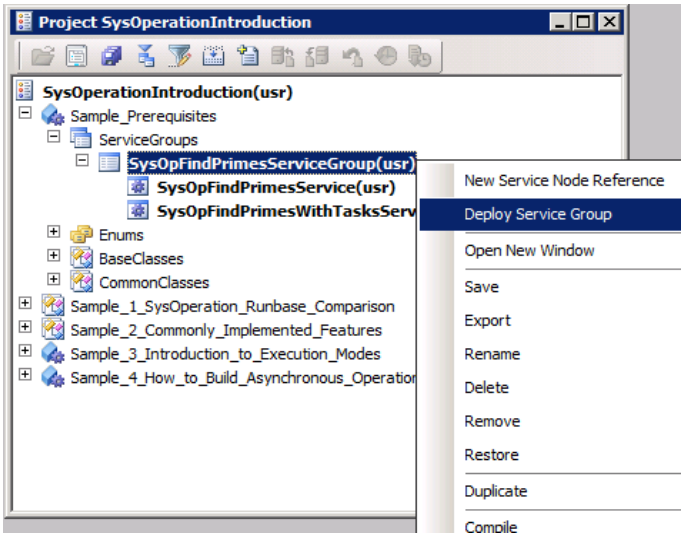
3. Compile the project, and make sure that there are no errors. (There are two warnings which will not the sample execution).



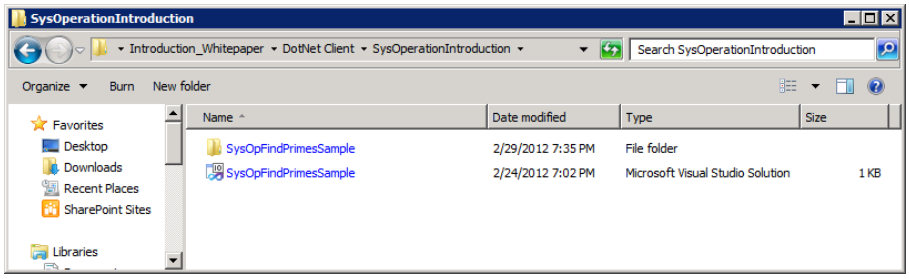
4. Generate CIL incrementally from the main toolbar in the developer workspace window. For more information about this topic, see the following article on MSDN: http://msdn.microsoft.com/en-us/library/gg839855.aspx.



5. Deploy the service group SysOpFindPrimesServiceGroup, which is located under the CommonItems\ServiceGroups group in the project. Deploying a service group generates a .NET assembly from X++ code and starts a service host that provides a service endpoint for all the service interfaces listed in the service group. For more information about service groups, see the following article on MSDN: http://msdn.microsoft.com/en-us/library/gg731906.aspx. Note that the services deployed in this step are required only for the .NET client sample and are not used in any of the other samples.



6. Open the .NET console application sample, and compile it to make sure that it has no errors.



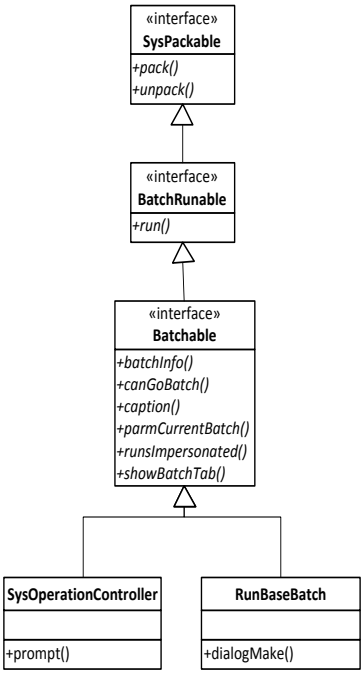
The console application is meant to work on the same machine as the Microsoft Dynamics AX server. If it is not on the same machine, open the app.config file in the solution, and update the **<client>** section to point to the Microsoft Dynamics AX server machine.

After these steps are completed, all the actions outlined in this paper can be duplicated by using the sample code.

Sample 1: Comparison of the SysOperation and RunBase frameworks

SysOperation and RunBase are frameworks geared toward building operations that can run via the batch server or interactively. In order for an operation to run via the batch server, it must meet these requirements:

- It must support parameter serialization via the SysPackable interface.
- It must support the standard **run()** method defined in the BatchRunable interface.
- It must support the batch server integration methods found in the Batchable interface.
- It must provide a mechanism to show input parameters with a user interface.

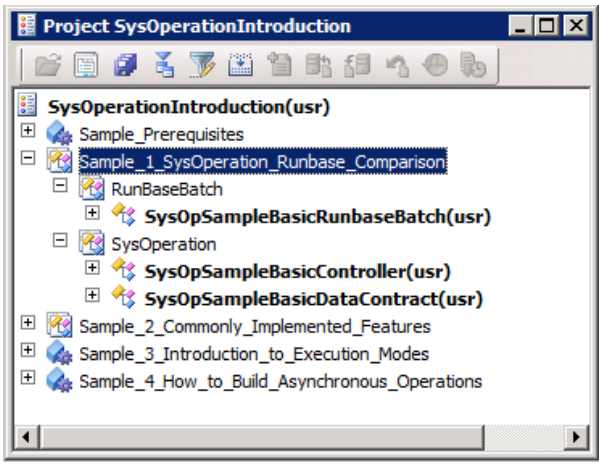


Currently in Microsoft Dynamics AX, all operations that must run via the batch server must derive from either the **SysOperationController** or the **RunBaseBatch** base class.

The following two samples illustrate the basic capabilities provided by the two frameworks:

- SysOpSampleBasicRunbaseBatch
- SysOpSampleBasicController

These can be found in the sample X++ project under the Sample_1_SysOperation_RunBase_Comparision group.



RunBase sample: SysOpSampleBasicRunbaseBatch

The simplest operation based on the **RunBaseBatch** base class has to implement 12 overrides. The purpose of this sample is simply to compare the RunBase and SysOperation frameworks. For full details of the RunBase framework, see the following article on MSDN: http://msdn.microsoft.com/en-us/library/aa863262.aspx.

Override	Description	Sample code

classDeclaration	• Derives from **RunBaseBatch**.
	• Declares variables for operation input parameters.
	• Declares variables for dialog box controls.
	• Declares a macro defining a list of variables that need to be serialized.
		`class SysOpSampleBasicRunbaseBatch extends RunBaseBatch`
		`{`
		`  str text;`
		`  int number;`
		`  DialogRunbase  dialog;`
		
		`  DialogField numberField;`
		`  DialogField textField;`
		
		`  #define.CurrentVersion(1)`
		
		`  #LOCALMACRO.CurrentList`
		`  text,`
		`  number`
		`  #ENDMACRO`
		`}`

dialog	Populates the dialog box created by the base class with controls needed to get user input. The initial values from the class member variables are used to initialize the controls. The type of each control is determined by the EDT identifier name.	`protected Object dialog()`
		`{`
		`  dialog = super();`
		
		`  textField = dialog.addFieldValue(IdentifierStr(Description255),`
		`  text,`
		`  'Text Property',`
		`  'Type some text here');`
		
		`  numberField = dialog.addFieldValue(IdentifierStr(Counter),`
		`  number,`
		`  'Number Property',`
		`  'Type some number here');`
		
		`  return dialog;`
		`}`

getFromDialog	Transfers the contents of dialog box controls to operation input parameters.	`public boolean getFromDialog()`
		`{`
		`  text = textField.value();`
		`  number = numberField.value();`
		
		`  return super();`
		`}`

putToDialog	Transfers the contents of operation input parameters to dialog box controls.	`protected void putToDialog()`
		`{`
		`  super();`
		
		`  textField.value(text);`
		`  numberField.value(number);`
		`}`

pack	Serializes operation input parameters.	`public container pack()`
		`{`
		`  return [#CurrentVersion, #CurrentList];`
		`}`

Override	Description	Sample code

unpack	Deserializes operation input parameters.	`public boolean unpack(container packedClass)`
		`{`
		`  Integer version = conPeek(packedClass,1);`
		`  switch (version)`
		`  {`
		`  case #CurrentVersion:`
		`  [version,#CurrentList] = packedClass;`
		`  break;`
		`  default:`
		`  return false;`
		`  }`
		`  return true;`
		`}`

run	Runs the operation. This sample prints the input parameters via the Infolog. It also prints the tier that the operation is running on and the runtime that is used for execution.	`public void run()`
		`{`
		`  if (xSession::isCLRSession())`
		`  {`
		`  info('Running in a CLR session.');`
		`  }`
		`  else`
		`  {`
		`  info('Running in an interpreter session.');`
		`  if (isRunningOnServer())`
		`  {`
		`  info('Running on the AOS.');`
		`  }`
		`  else`
		`  {`
		`  info('Running on the Client.');`
		`  }`
		`  }`
		`  info(strFmt('SysOpSampleBasicRunbaseBatch: %1, %2', this.parmNumber(), this.parmText()));`
		`}`

description	A static description for the operation. This description is used as the default value for the caption shown in batch and the operation user interface.	`public static ClassDescription description()`
		`{`
		`  return 'Basic RunBaseBatch Sample';`
		`}`

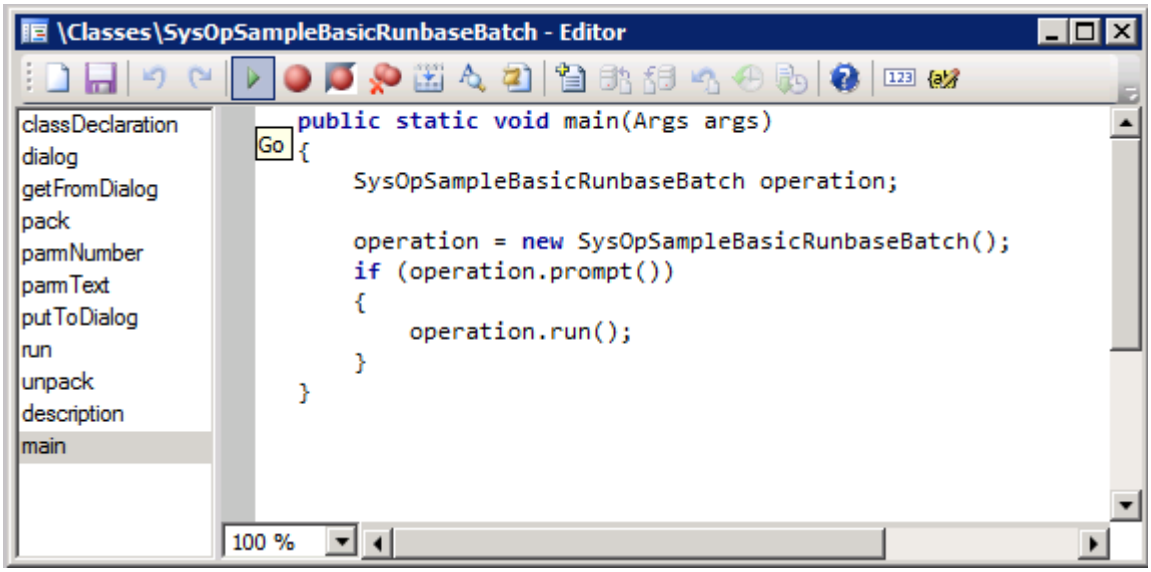
main	The main interaction code for the operation. This code prompts the user for input, and then runs the operation or adds it to the batch queue.	`public static void main(Args args)`
		`{`
		`  SysOpSampleBasicRunbaseBatch operation;`
		`  operation = new SysOpSampleBasicRunbaseBatch();`
		`  if (operation.prompt())`
		`  {`
		`  operation.run();`
		`  }`
		`}`

Override	Description	Sample code

parmNumber	Optional. It is a Microsoft Dynamics AX best practice to expose operation parameters with the property pattern for better testability and for access to class member variables outside the class.	`public int parmNumber(int _number = number)`
		`{`
		`  number = _number;`
		`  return number;`
		`}`

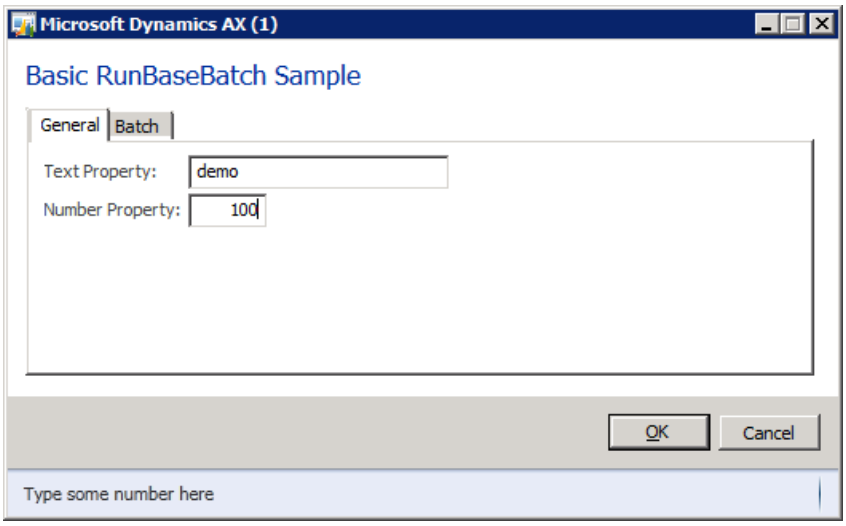
parmText	Optional. It is a best practice to expose operation parameters with the property pattern.	`public str parmText(str _text = text)`
		`{`
		`  text = _text;`
		`  return text;`
		`}`

After it is implemented, the operation can be run by using the **Go** button on the code editor toolbar.

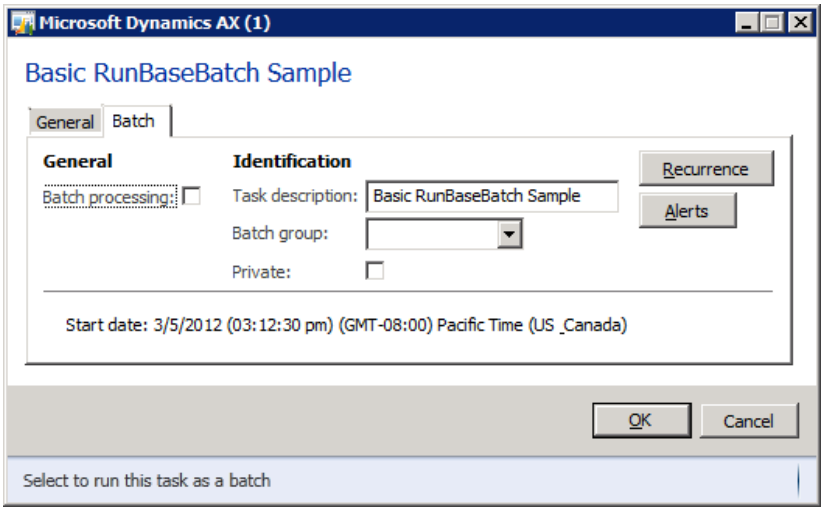


If an X++ class implements the main operation, it is automatically called by the code editor. The sample's main operation will prompt the user for input for the operation when **operation.prompt()** is called. If the prompt returns **true**, **main** calls **operation.run()** directly. If the prompt returns **false**, the user either canceled the operation or scheduled it via batch.

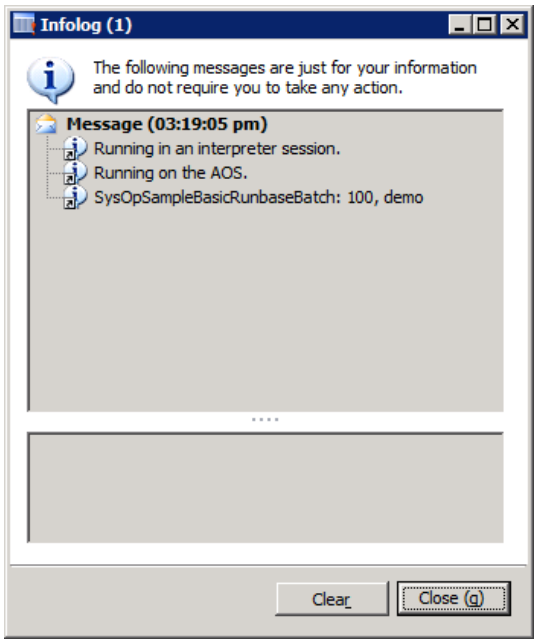
To run the operation interactively, enter data on the **General** tab of the operation user interface.



Make sure that the **Batch processing** check box is cleared on the **Batch** tab.

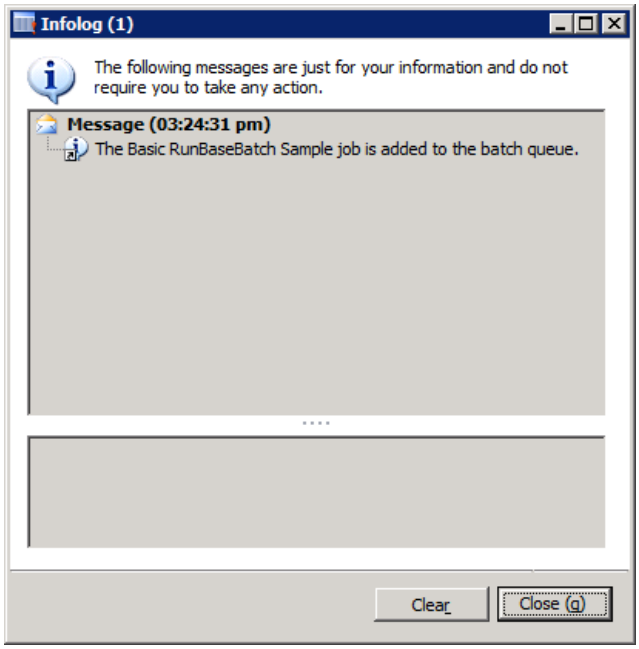


Clicking **OK** will run the operation and print the following output to the Infolog window.

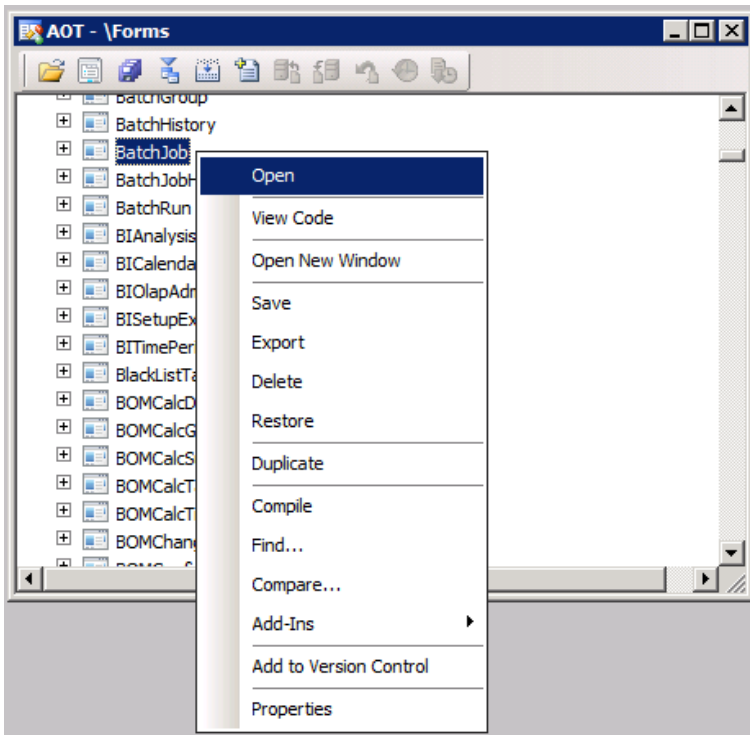


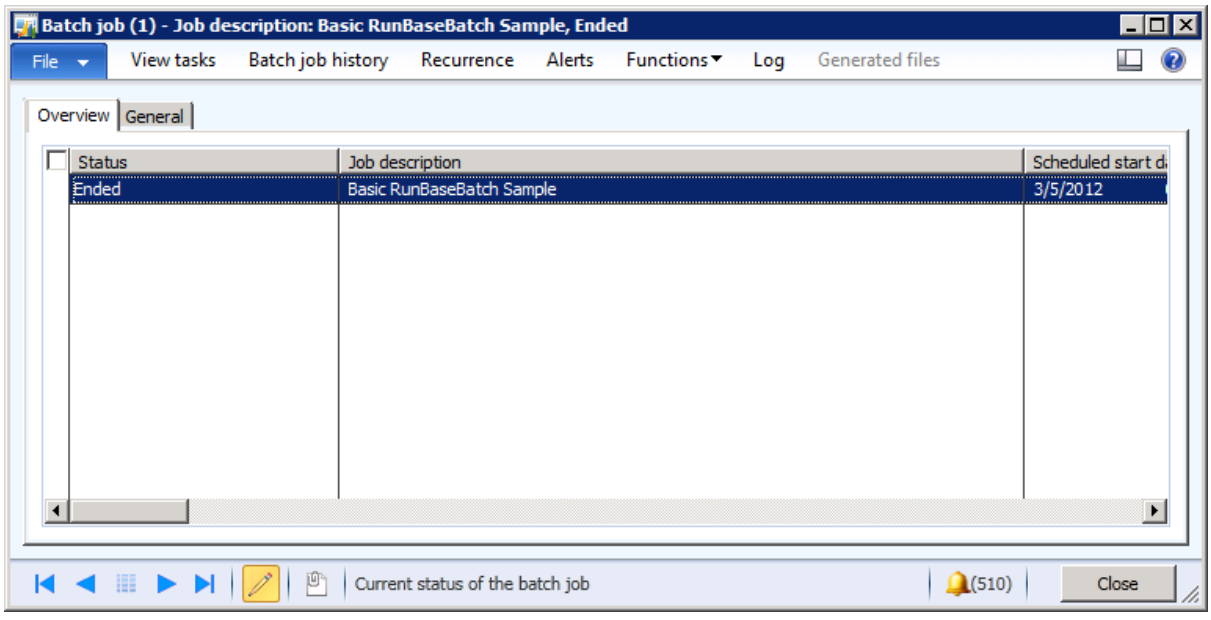
The Infolog messages show that the operation ran on the server, because the sample class is marked to run on the server. The operation ran via the X++ interpreter, which is the default for X++ code.

If you repeat the previous steps but select the **Batch processing** check box on the **Batch** tab, the operation will to run via the batch server. When the **Batch processing** check box is selected, the following Infolog message is shown, indicating that the operation has been added to the batch queue.



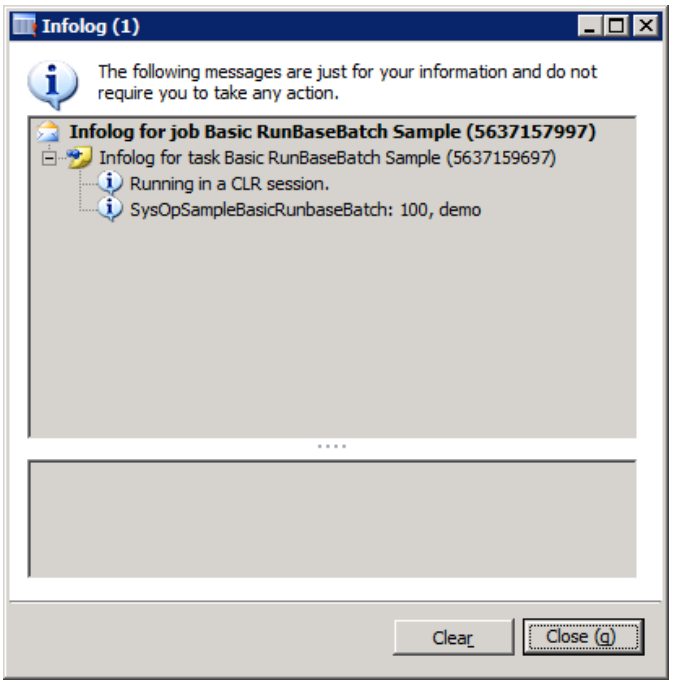
The operation may take up to a minute to get scheduled. After waiting for about a minute, open the **Batch job** form from the Application Object Tree (AOT).





Repeatedly update the form by pressing the F5 key, until the job entry shows that the job has ended. Sorting by the **Scheduled start date/time** column may help you find the operation if there are many job entries in the grid. After you find the correct job, select it, and then click **Log** on the toolbar.

Clicking **Log** opens an Infolog window indicating that the operation ran in a CLR session, which is the batch server execution environment.



In summary, this sample shows the minimum overrides needed to create an operation that can run either interactively or via the batch server by using the **RunBaseBatch** base class.

SysOperation sample: SysOpSampleBasicController

The purpose of the SysOperation framework is to provide the same capabilities as the RunBase framework but with base implementations for common overrides. The SysOperation framework handles basic user interface creation, parameter serialization, and routing to the CLR execution environment. The following table of overrides shows the code needed to match the functionality demonstrated for the RunBase-based sample in the previous section.

The SysOperation sample contains two classes: a controller class named **SysOpSampleBasicController** and a data contract class named **SysOpSampleBasicDataContract**.

SysOpSampleBasicController should derive from **SysOperationServiceController**, which provides all the base functionality for building operations; however, there are a few issues with that class as it is shipped with Microsoft Dynamics AX 2012, and these will be addressed in a future service pack. In the meantime, to work around the issues, a new common class, **SysOperationSampleBaseController**, is provided. Details of the issues worked around will be discussed at the end of this paper.

The following table shows the code needed to provide equivalent functionality to the RunBase sample.

Override	Description	Sample code
SysOpSampleBasicController		

classDeclaration	Derives from the framework base class **SysOpSampleBaseController**.
	Normally the operation should derive from the SysOperationServiceController class. The sample base class provides a few fixes for issues in that class.
		`class SysOpSampleBasicController extends SysOpSampleBaseController`
		`{`
		`}`

Override	Description	Sample code

new	Identifies the class and method for the operation. In the sample, this points to a method on the controller class; however, in general, it can be any class method.
	The framework will reflect on this class/method to automatically provide the user interface and parameter serialization.
		`void new()`
		`{`
		`  super();`
		
		`  this.parmClassName(`
		`classStr(SysOpSampleBasicController));`
		`  this.parmMethodName(`
		`methodStr(SysOpSampleBasicController,`
		`  showTextInInfolog));`
		
		`  this.parmDialogCaption(`
		`  'Basic SysOperation Sample');`
		`}`

Dialog	Base functionality implemented by the framework.	
getFromDialog		
putToDialog		
pack		
unpack		
run	Implemented by the base framework. Handles marshaling execution to a CLR session.	

showTextInInfolog	Prints the input parameters via the Infolog. Also prints the tier that the operation is running on and the runtime that is used for execution.	`public void showTextInInfolog(SysOpSampleBasicDataContract data)`
		`{`
		`  if (xSession::isCLRSession())`
		`  {`
		`  info('Running in a CLR session.');`
		`  }`
		`  else`
		`  {`
		`  info('Running in an interpreter session.');`
		`  if (isRunningOnServer())`
		`  {`
		`  info('Running on the AOS.');`
		`  }`
		`  else`
		`  {`
		`  info('Running on the Client.');`
		`  }`
		`  }`
		
		`info(strFmt('SysOpSampleBasicController: %1, %2', data.parmNumber(), data.parmText()));`
		`}`

caption	A description for the operation. This description is used as the default value for the caption shown in batch and the operation user interface.	`public ClassDescription caption()`
		`{`
		`  return 'Basic SysOperation Sample';`
		`}`

Override	Description	Sample code

main	The main interaction code for the operation. This code prompts the user for input, and then runs the operation or adds it to the batch queue.	`public static void main(Args args)`
		`{`
		`  SysOpSampleBasicController operation;`
		`  operation = new SysOpSampleBasicController();`
		`  operation.startOperation();`
		`}`

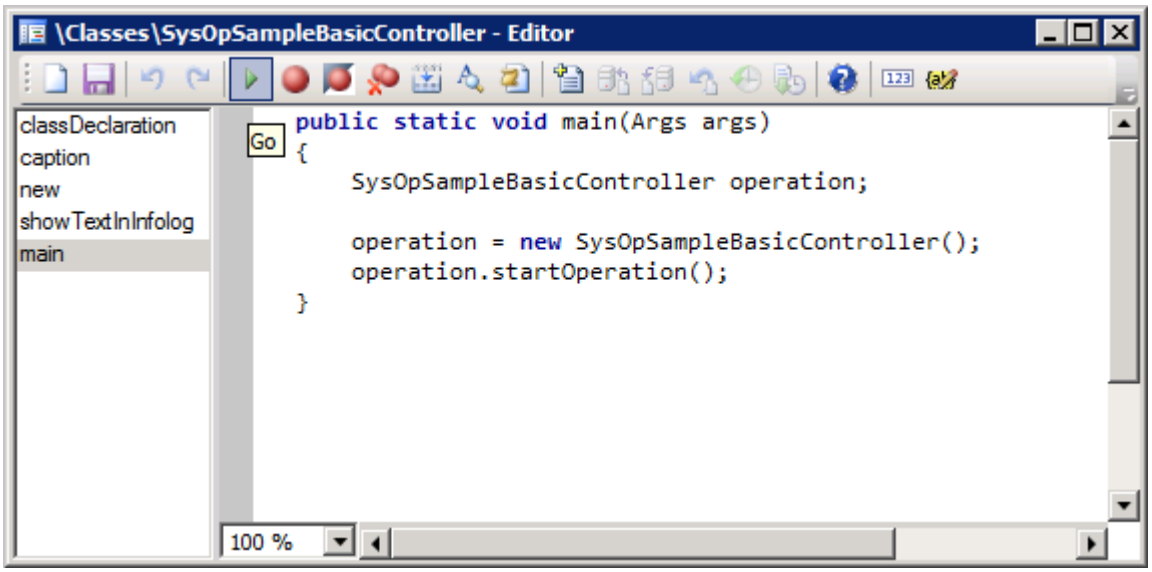
SysOpSampleBasicDataContract		

classDeclaration	The data contract attribute is used by the base framework to reflect on the operation.	`[DataContractAttribute]`
		`class SysOpSampleBasicDataContract`
		`{`
		`  str text;`
		`  int number;`
		`}`

parmNumber	The data member attribute identifies this property method as part of the data contract. The label, help text, and display order attributes provide hints for user interface creation.	`[DataMemberAttribute,`
		`SysOperationLabelAttribute('Number Property'),`
		`SysOperationHelpTextAttribute('Type some number >= 0'),`
		`SysOperationDisplayOrderAttribute('2')]`
		`public int parmNumber(int _number = number)`
		`{`
		`  number = _number;`
		`  return number;`
		`}`

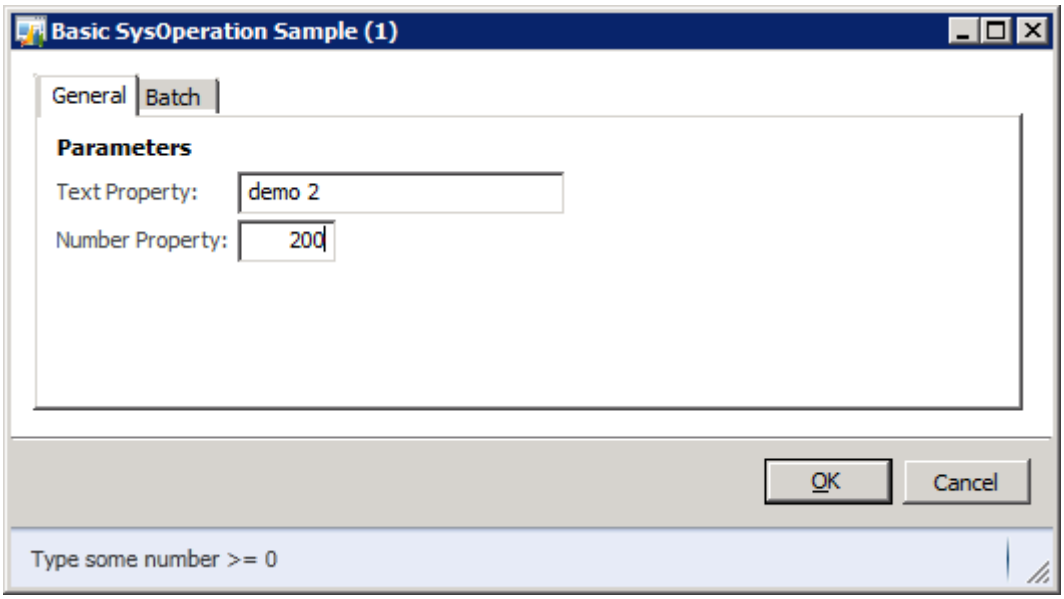
parmText	The data member attribute identifies this property method as part of the data contract. The label, help text, and display order attributes provide hints for user interface creation.	`[DataMemberAttribute,`
		`SysOperationLabelAttribute('Text Property'),`
		`SysOperationHelpTextAttribute('Type some text'),`
		`SysOperationDisplayOrderAttribute('1')]`
		`public Description255 parmText(str _text = text)`
		`{`
		`  text = _text;`
		`  return text;`
		`}`

As in the RunBase sample, the operation can be run by using the **Go** button on the code editor toolbar.

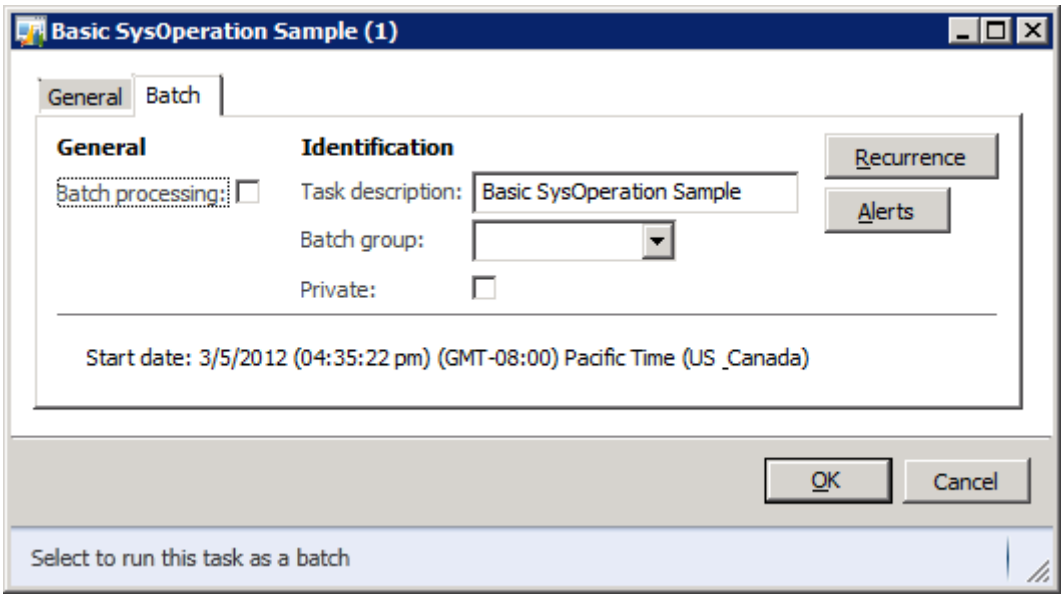


The main class calls **operation.startOperation()**, which handles running the operation synchronously or adding it to the batch queue. Although **operation.run()** can also be called, this should be done only if the data contract has been programmatically filled out. The **startOperation** method invokes the user interface for the operation, and then calls **run**.

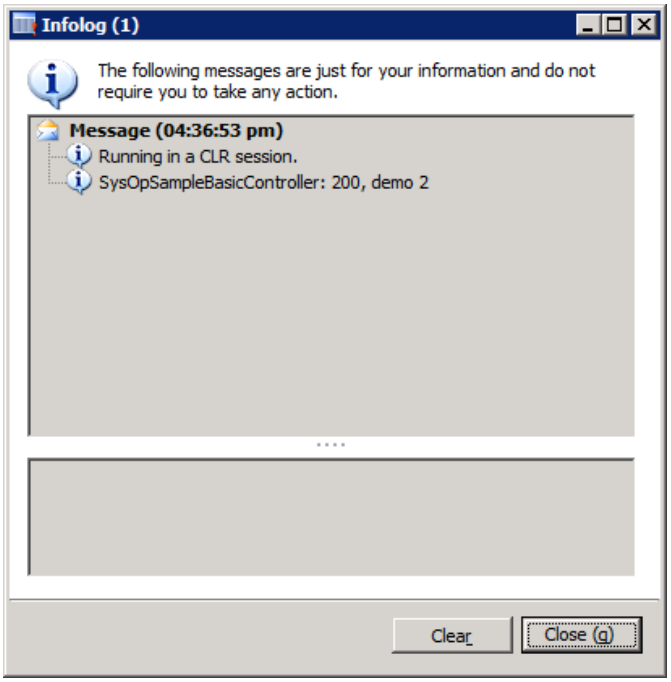
To run the operation interactively, enter data on the **General** tab of the operation user interface. The user interface created by the framework is very similar to the one created in the RunBase sample.



Make sure that the **Batch processing** check box is cleared on the **Batch** tab.

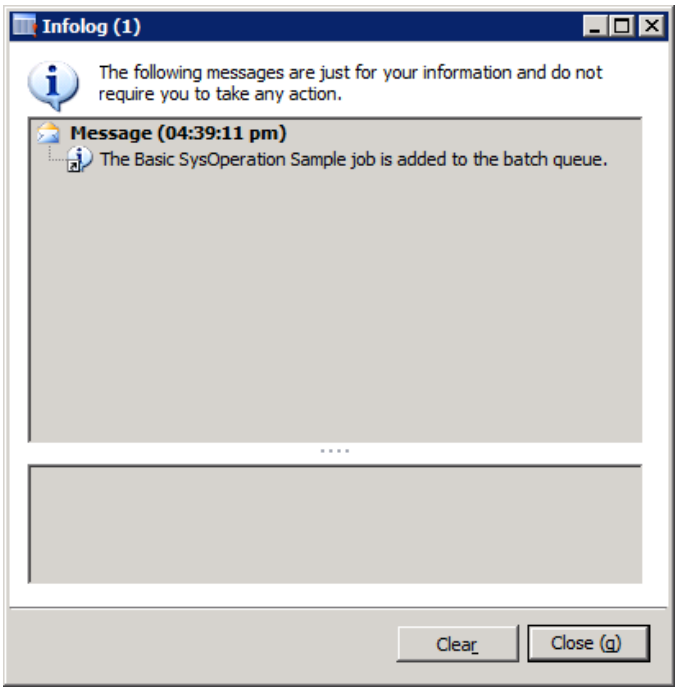


Clicking **OK** will run the operation and print the following output to the Infolog window.

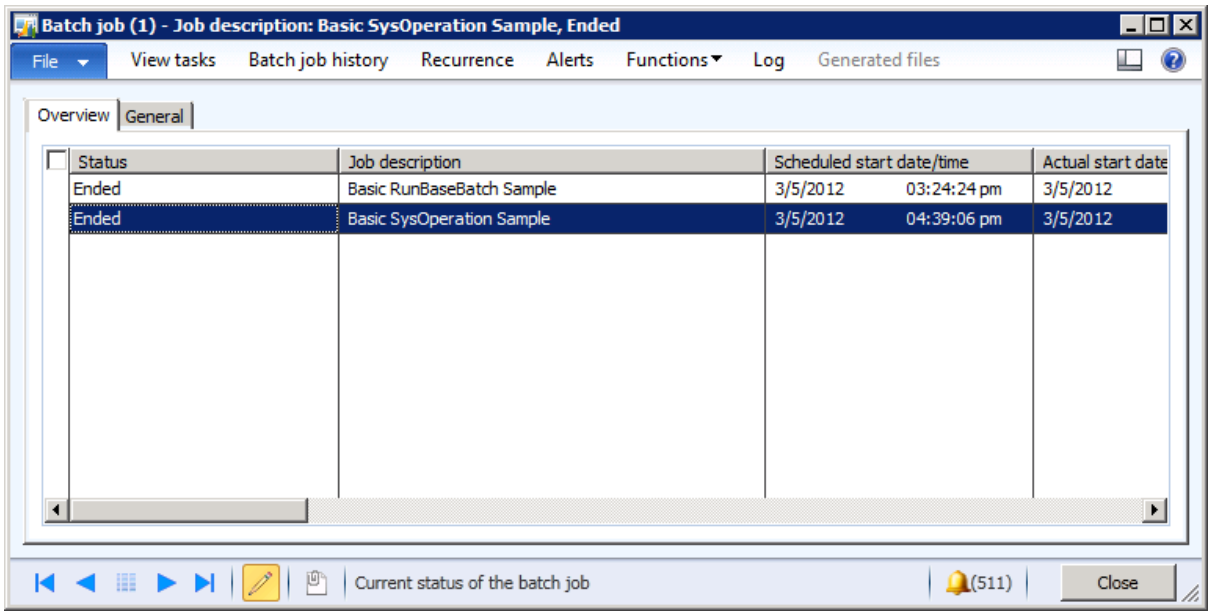


The Infolog messages show that, unlike in the RunBase sample, the operation ran in a CLR session on the server.

If you repeat the previous steps but select the **Batch processing** check box on the **Batch** tab, the operation will run via batch, just as in the RunBase sample.

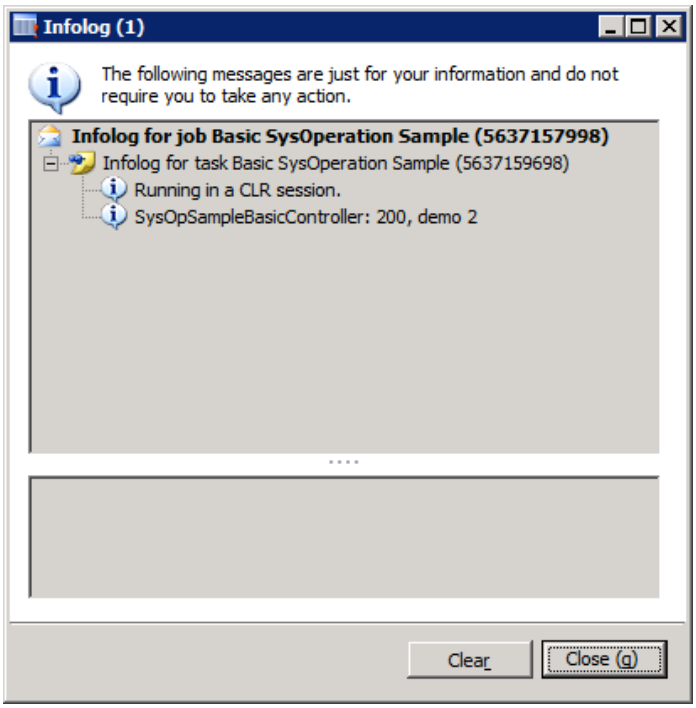


The operation may take up to a minute to get scheduled. After waiting for about a minute, open the **Batch job** form from the AOT, as in the RunBase sample.



Repeatedly update the form by pressing the F5 key, until the job entry shows that the job has ended. Sorting by the **Scheduled start date/time** column may help you find the operation if there are many jobs entries in the grid. After you find the correct job, select it, and then click **Log** on the toolbar.

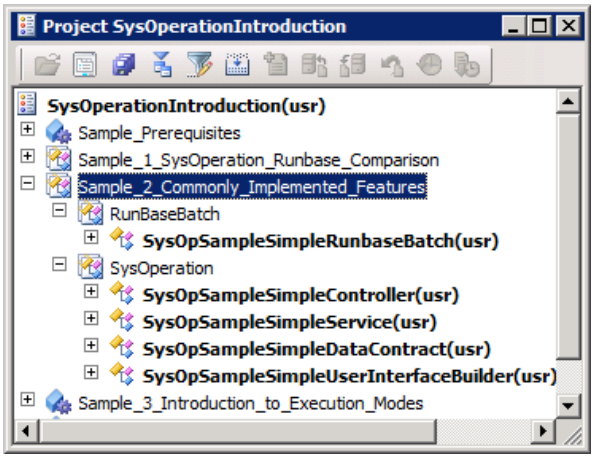
Clicking **Log** opens an Infolog window indicating that the operation ran in a CLR session, which is the batch server execution environment.



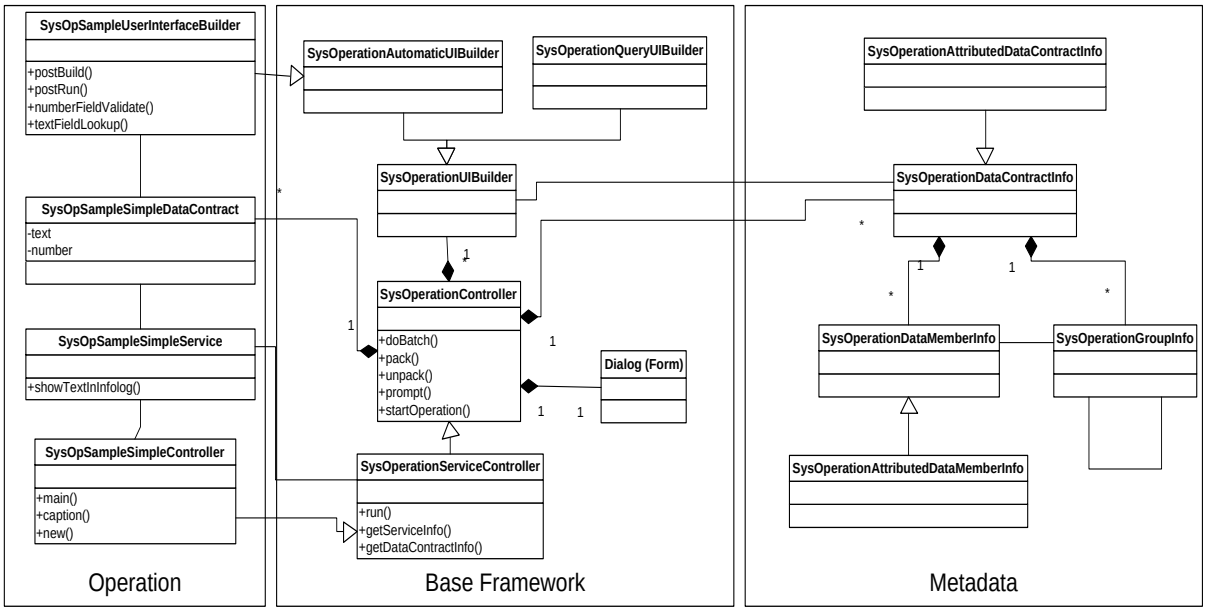
This sample showed that the SysOperation framework can provide the same basic functionality as the RunBase framework. In addition, it provides implementation for common RunBase overrides, such as parameter serialization and user interface creation. It also provides the capability to route synchronous operations to a CLR session.

Sample 2: Demonstration of commonly implemented features in SysOperation and RunBase

Basic samples used for demonstration rarely meet the requirements of real-life scenarios. The group named Sample_2_Commonly_Implemented_Features in the X++ sample project, illustrates common functionality associated with user interfaces in Microsoft Dynamics AX: lookups and validation. The samples illustrate the same functionality implemented in both frameworks to facilitate comparison.



The SysOperation sample, SysOpSampleSimpleController, is factored differently from the first sample, SysOpSampleBasicController. In this sample, there are four classes: the controller, the service operation, the data contract, and the user interface builder. The following figure outlines the classes in relation to the base framework classes.



The service and data contract classes define the operation. The derived controller class provides the main entry point and overrides the **new()** method to associate the operation classes with the controller. The base controller reflects on the operation and constructs metadata classes that define the operation. The base class **SysOperationAutomaticUIBuilder** uses the metadata derived from the operation to create the user interface. In the sample, there is a derived user interface builder called **SysOpSampleSimpleUserInterfaceBuilder**. This overrides the **postBuild()** and **postRun()** overrides on the base builder to subscribe to form control events related to validation and lookup.

The system uses **SysOperationContractProcessingAttribute** to associate the custom user interface builder with the data contract.

```
[DataContractAttribute,
SysOperationContractProcessingAttribute(classStr(SysOpSampleSimpleUserInterfaceBuilder))]
class SysOpSampleSimpleDataContract
{
    str text;
    int number;
}
```

If this attribute is not present, the default builder, **SysOperationAutomaticUIBuilder**, is used. As an experiment, comment out the attribute in the preceding code, and then run the operation to see the differences.

The **postBuild()** override in the custom user interface builder is where the form control metadata needs to be modified before the controls are instantiated. The framework maintains an association between controls and data contracts in a map that can be accessed via the **this.bindInfo()** method. The map is keyed by the name of the property in the data contract.

```
public void postBuild()
{
    super();

    // get references to dialog controls after creation
    numberField = this.bindInfo().getDialogField(this.dataContractObject(),
```

```
        methodStr(SysOpSampleSimpleDataContract, parmNumber));
    textField = this.bindInfo().getDialogField(this.dataContractObject(),
        methodStr(SysOpSampleSimpleDataContract, parmText));
    // change text field metadata to add lookup
    textField.lookupButton(#lookupAlways);

}
```

The **postRun()** override in the custom user interface builder is where the form control events are subscribed to. The subscriptions must be added to the controls after they have been instantiated.

```
public void postRun()
{
    super();

    // register overrides for form control events
    numberField.registerOverrideMethod(methodstr(FormIntControl, validate),
        methodstr(SysOpSampleSimpleUserInterfaceBuilder, numberFieldValidate), this);
    textField.registerOverrideMethod(methodstr(FormStringControl, lookup),
        methodstr(SysOpSampleSimpleUserInterfaceBuilder, textFieldLookup), this);

}
```

The **registerOverRideMethod** method on the controls is a run-time equivalent to the control overrides used in normal forms. If you use an override method in a standard Microsoft® MorphX® form, you can use the same method override in a dynamic form by using this mechanism. Note that both the RunBase and SysOperation frameworks allow the use of modeled forms as the operation user interface. The SysOperation framework provides the override **SysOperationController.templateForm()** for that purpose, however, this topic is outside the scope of this white paper.

The samples in this section show how the user interface for the operation can use many of the same features that are available in the normal form programming model. Control overrides fire run-time events that can be subscribed to. The SysOperation version of the sample shows how the different aspects of the operation can be factored into separate classes.

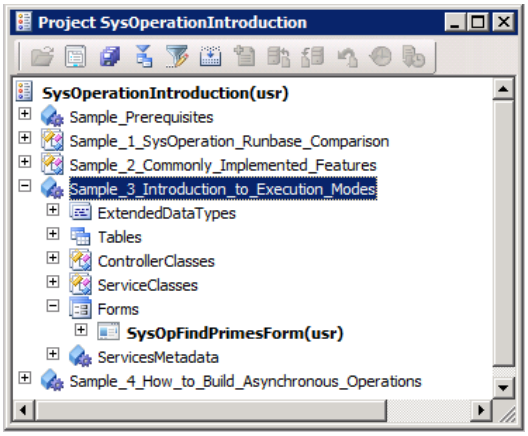
To show that everything is possible with code, the RunBase sample is modified so that it marshals its interactive execution into a CLR session, in the same way that the SysOperation framework does. This illustrates the design principle that drove the SysOperation framework: move as much of the boilerplate code as possible into the base classes.

```
private static server void showTextInInfolog(container packedRunBase)
{
    SysOpSampleSimpleRunbaseBatch thisClass;

    // If not in a CLR session then marshal over. If already in a CLR session
    // then execute the logic for the operation
    if (!xSession::isCLRSession())
    {
        new XppILExecutePermission().assert();
        SysDictClass::invokeStaticMethodIL(classStr(SysOpSampleSimpleRunbaseBatch),
                                           staticMethodStr(SysOpSampleSimpleRunbaseBatch,
                                               showTextInInfolog),
                                           packedRunBase);
        // exit call executed in CLR session.
        return;
    }

    thisClass = new SysOpSampleSimpleRunbaseBatch();
    if (!thisClass.unpack(packedRunBase))
    {
        throw AifFault::fault('SysOpSampleSimpleRunbaseBatch unpack error', 'unpackError');
```

```
    }

    if (xSession::isCLRSession())
    {
        info('Running in a CLR session.');
    }
    else
    {
        info('Running in an interpreter session.');
        if (isRunningOnServer())
        {
            info('Running on the AOS.');
        }
        else
        {
            info('Running on the Client.');
        }
    }

    info(strFmt('SysOpSampleSimpleRunbaseBatch: %1, %2',
        thisClass.parmNumber(), thisClass.parmText()));
}
```

Sample 3: Introduction to SysOperation execution modes

This sample provides a basic introduction to the reliable asynchronous execution mode that is available in the batch server.



Execution modes overview

The concept of sessions in Microsoft Dynamics AX is important to understanding execution modes. A session is defined as the runtime context under which the X++ code is executed. The session defines the current user, permissions, and the in-memory state of the various X++ and kernel objects that the user is interacting with. Sessions in Microsoft Dynamics AX are single-threaded: only a single task can be executed in a session at a given time. Understanding this constraint is especially important when executing in a session that is initiated from the Dynamics AX client. A lengthy operation in a session that is initiated from the client can block the user interface and cause the client to stop responding.

The execution modes for the SysOperation framework are designed to provide different options for managing the single-threaded constraint that is associated with Microsoft Dynamics AX sessions.

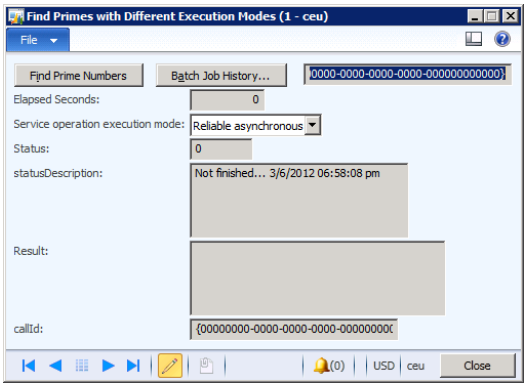
The following table lists the four execution modes that are defined by the SysOperation framework enum **SysOperationExecutionMode** and describes their operations.

Execution Mode	Description	Use
Reliable asynchronous	Reliable asynchronous operations use the batch server's session for execution. The call is queued to the empty batch queue by default, but the appropriate queue can be specified by using a property on the operation. It is possible to track the execution of reliable asynchronous calls by tracking the associated batch execution history. It is also possible to make the initiation of the reliable asynchronous call part of a business process transaction. Reliable asynchronous calls can make use of the batch server's parallel execution mechanism. IMPORTANT: For these reasons, reliable asynchronous execution is the recommended mechanism for executing lengthy calls.	Running operations in this mode is equivalent to running them on the batch server, with the additional behavior that the jobs are automatically deleted after they are completed, whether they were successfully completed or not. However, the job history is persisted in the system. This pattern is provided to facilitate building operations that use the batch server runtime, but that do not rely on the batch server administration features. These jobs only temporarily show up in the **Batch job** form and can be filtered out completely by setting a filter in the **BatchJob.RuntimeJob** field. To implement this behavior, the batch header exposes a property named **parmRuntimeJob**, which is set to **True** by the SysOperation framework to select this behavior. For more information about batch jobs and tasks, see the following article on MSDN: http://technet.microsoft.com/en-us/library/dd309586.aspx.
Scheduled batch	Scheduled batch mode uses the traditional batch server execution mechanism. This execution mode differs from the reliable asynchronous mode only in the way batch jobs persist in the system.	You use this mode to run batch jobs in the asynchronous, server-based, batch processing environment. Unlike reliable asynchronous calls, scheduled batch jobs are persisted in the system until a user manually deletes them.
Synchronous	Synchronous calls are always initiated in the caller's session. The calls are always marshaled to the server and executed in intermediate language (IL). (There is a mechanism to opt out of having the calls always marshaled to the server. For more information, see http://msdn.microsoft.com/en-us/library/sysoperationservicecontroller.executeoperationwithrunas.aspx). If the class being called is registered as a service in the **AxClient** service group, then a Windows Communication Foundation (WCF) service proxy is used to marshal the synchronous call to the server. If the class being called is not registered as a service in the **AxClient** service group then the call is marshaled to the server using the operation's pack/unpack mechanism and is executed in IL using the runAs mechanism. If the call is initiated on the server, then it is marshaled to IL, if needed, before the	You should use the synchronous execution method when the operation is not lengthy or when it is initiated from a batch session. Results may be obtained by using the operation's result parameter.

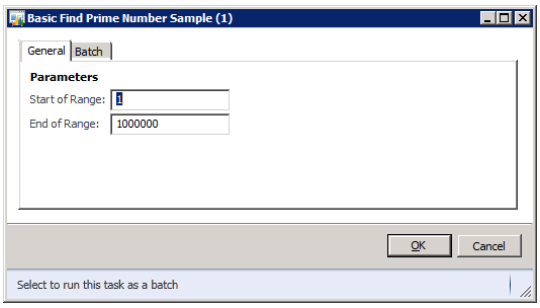
	service method is called.	
Asynchronous	Asynchronous operations are very similar to synchronous operations except that they are executed by using the WCF asynchronous service call mechanism. Asynchronous calls only run asynchronously if they are initiated from the desktop client session and if the called service is registered in the **AxClient** service group. In all other cases, the calls are executed synchronously. In all cases the caller can get results by using the SysOperationServiceController.operationReturnValue Method in the afterOperation override method. For more information, see http://msdn.microsoft.com/en-us/library/sysoperationservicecontroller.afteroperation(v=ax.60).aspx and http://msdn.microsoft.com/en-us/library/sysoperationservicecontroller.operationreturnvalue.aspx. NOTE: Adding a service class that is used as part of a SysOperation to the **AxClient** service group is not usually required. The SysOperation framework always marshals the data contract to the server by value, whether or not the service class is published as a service. The Asynchronous execution mode is the only mode that explicitly requires **AxClient** server group registration.	Asynchronous calls are useful for running lengthy operations from the desktop client, where durability is not important. The caller must guarantee that the call is initiated from the desktop client to get the benefit of asynchronous execution.

Sample overview

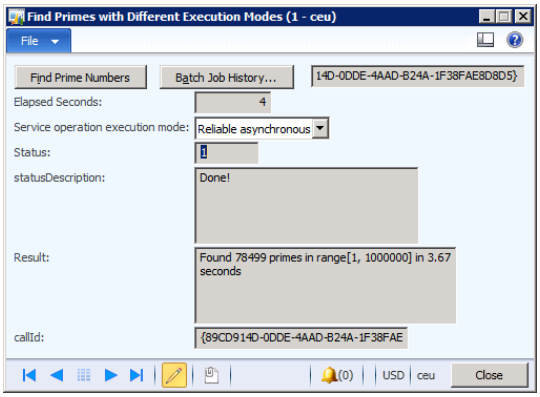
1. Open the form, and make sure that the **Service operation execution mode** field is set to **Reliable asynchronous** (which is the default value).



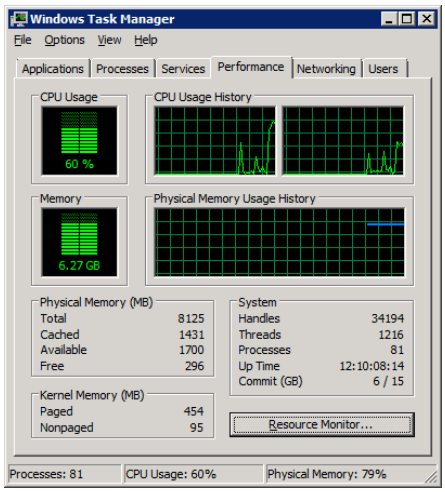
2. Click **Find Prime Numbers**, and then enter the operation input parameters. Enter the range **[1,1000000]**.



3. Click **OK** to start the operation. The form will poll for changes by using the **element.settimeout()** mechanism. The operation should be completed in a few seconds. Notice that the client is not frozen while the operation is running.



If you have multiple processors in your machine, note that the full power of all the processors **is not** used in the computation. There is only one thread.

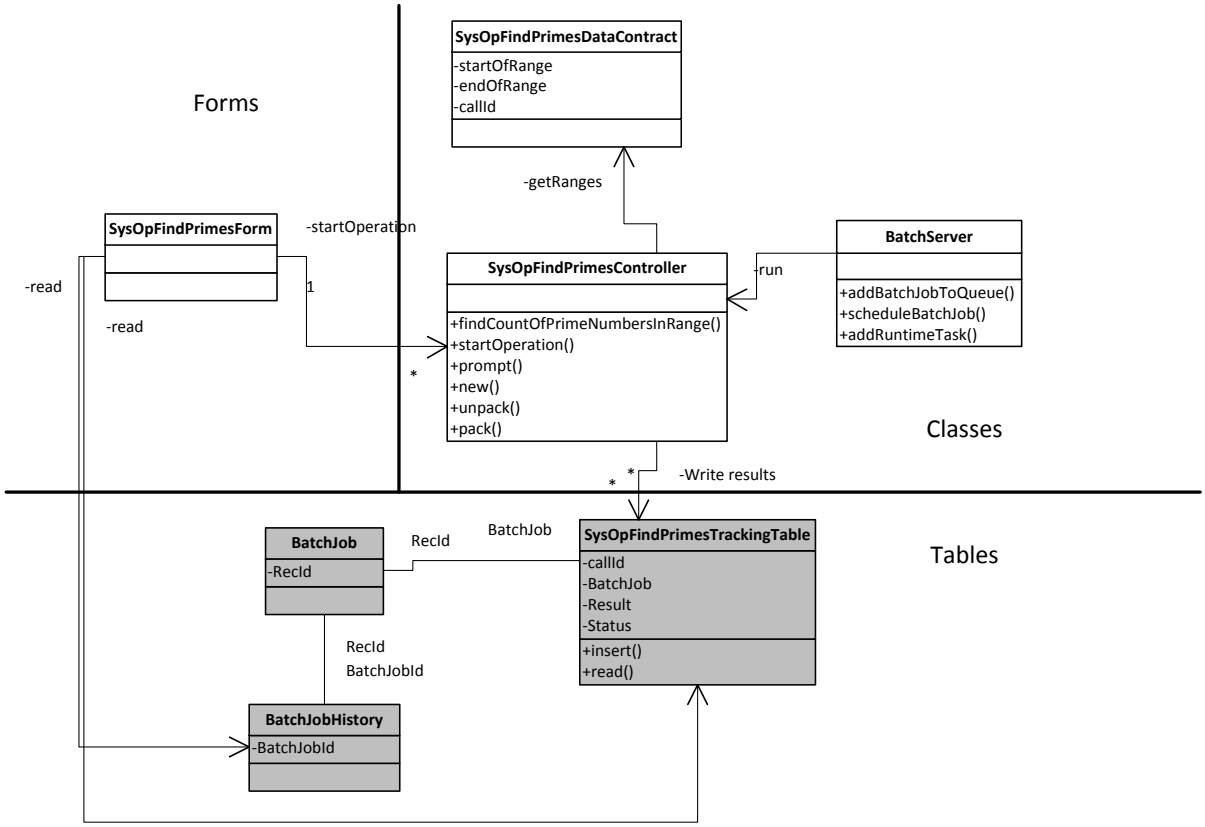


4. Try the operation again with larger ranges, to see how the application behaves.
5. Change the **Service operation execution mode** field to **Synchronous**, and try the operation again. If a large range is entered, the client will freeze while the operation is running.

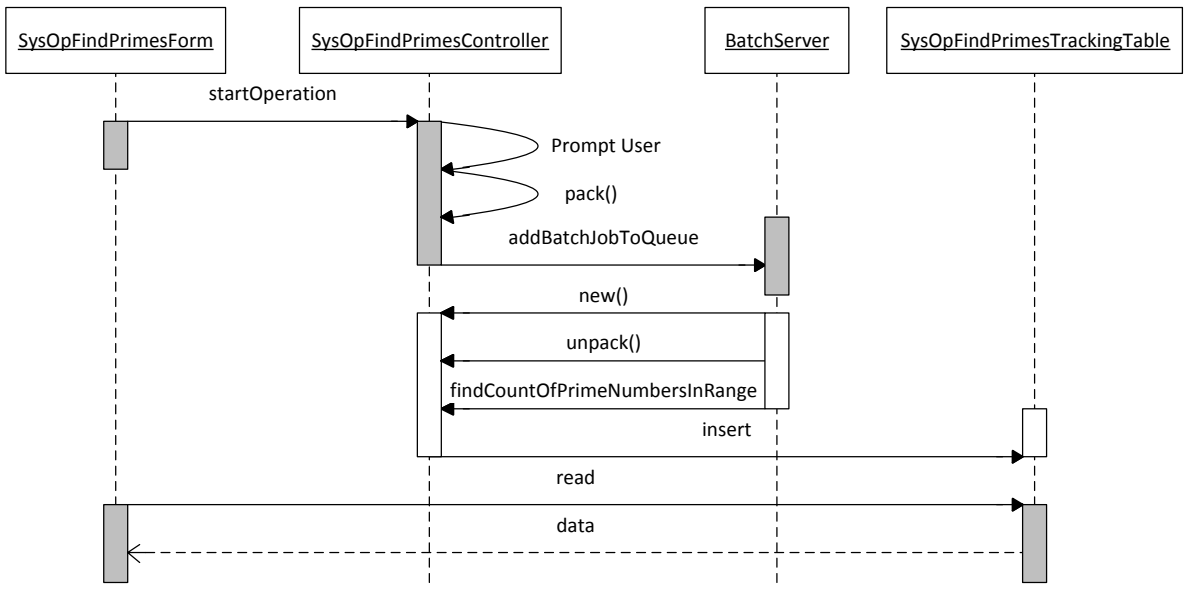
6. Change the **Service operation execution mode** field to **Scheduled Batch**, and try the operation again. In the **Batch job** form, in the **AOT Forms** node, view the difference between **Reliable Asynchronous** transient jobs and **Scheduled Batch** normal jobs. Transient or run-time jobs are automatically deleted on completion.

Architecture and code

Class diagram



Sequence diagram



As the preceding figures show, **SysOpFindPrimesForm** creates an instance of **SysOpFindPrimesController** and calls **startOperation** on it. The controller displays the user interface needed to get operation parameters and starts the operation. The operation updates SysOpFindPrimesTrackingTable, which **SysOpFindPrimesForm** polls to get the results.

The form starts the operation.

SysOpFindPrimesForm

```
void clicked()
{
    guid id;
    SysOperationStartResult result;
    Args args;

    elapsedSeconds.value(0);

    // Pass the execution mode to the controller from the combo box
    args = new args();
    args.parmEnumType(enumNum(SysOperationExecutionMode));
    args.parmEnum(operationExecutionMode.selection());

    [result, id] = SysOpFindPrimesController::main(args);

    // controller's main function passes the operation id back
    // the form uses this to poll for results.
    if (result == SysOperationStartResult::AddedToBatchQueue ||
        result == SysOperationStartResult::Started)
    {
        callId.value(id);

        initialTicks = WinAPI::getTickCount();

        element.recurringRefresh();
    }

    if (result == SysOperationStartResult::AddedToBatchQueue)
    {
        // Run the batch server ping operation in a CLR session
        // the controller will marshall it over. This will cause
        // the batch server to poll right away if it is idle.
        new SysOpSampleBatchServerPingController().run();
    }
}
```

Note the use of **SysOperationBatchServerPingController** in the preceding code. This notifies the batch server that there is additional work to do. The API simply sets a flag that is normally set by the batch server to process new tasks created during job execution. Setting the flag is simply an indication of pending work, it adds no overhead to the batch server runtime.

When the controller is run in **Reliable Asynchronous** mode, it adds itself as a batch job. When the batch server schedules the job, it creates the controller, hydrates it by calling **unpack()**, and calls the operation logic. The operation logic updates the result table.

SysOpFindPrimesController

```
public void findCountOfPrimeNumbersInRange(SysOpFindPrimesDataContract range)
{
    int64 i, primes;
    SysOpFindPrimesTrackingTable logTable;
    RefRecId batchJobId;
```

```
    int ticks;

    if (this.isInBatch())
    {
        batchJobId = BatchHeader::getCurrentBatchHeader().parmBatchHeaderId();
    }

    ttsBegin;
    logTable.clear();
    logTable.BatchJob = batchJobId;
    logTable.Status = 0;
    logTable.callId = range.parmCallId();
    logTable.Result = strFmt('Finding primes in range[%1, %2]', range.parmStartOfRange(),
                             range.parmEndOfRange());
    logTable.insert();
    ttsCommit;

    ticks = WinAPIServer::getTickCount();
    primes = 0;
    for (i = range.parmStartOfRange(); i <= range.parmEndOfRange(); i++)
    {
        if (SysOpFindPrimesController::isPrime(i))
        {
            primes++;
        }
    }
    ticks = WinAPIServer::getTickCount() - ticks;

    ttsBegin;
    logTable.Status = 1;
    logTable.callId = range.parmCallId();
    logTable.Result = strFmt('Found %3 primes in range[%1, %2] in %4 seconds',
                      range.parmStartOfRange(), range.parmEndOfRange(), primes, ticks / 1000);
    logTable.update();
    ttsCommit;

}
```

The form polls the table for the results.

SysOpFindPrimesForm

```
public void refresh()
{
    int seconds;

    // Poll the result table using the operation id
    callIdRange.value(callId.valueStr());

    // execute the query on the datasource
    TrackingTable_ds.executeQuery();

    // update the elapsed seconds
    seconds = (WinAPI::getTickCount() - initialTicks) / 1000;
    elapsedSeconds.value(seconds);
}
```

The form periodically polls the table until Status != 0.

SysOpFindPrimesForm

```
public void recurringRefresh()
{
    element.refresh();

    if (TrackingTable.Status == 0)
    {
        element.setTimeOut(identifierStr(recurringRefresh), 1000);
    }
}
```

Initialize the default values in a data contract.

Derive from **SysOperationInitializable**.

```
[DataContractAttribute]
class SysOpFindPrimesDataContract implements SysOperationInitializable
```

Override the initialize method.

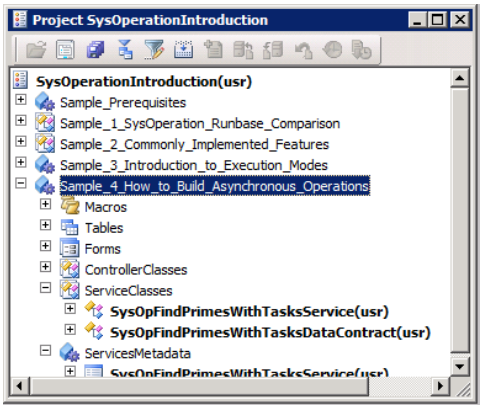
```
// initialize is called if the value
// is not found in the syslastvalue table
public void initialize()
{
    startOfRange = 1;
    endOfRange = 1000000;
}
```

Note that, in this sample, the **SysOpFindPrimesService** class is not used. It will be used later, from the .NET client, to invoke the controller discussed here.

Sample 4: How to build asynchronous operations with the SysOperation framework

The previous sample introduced architecture for building asynchronous operations by using transient batch jobs or run-time jobs. The final sample, AsynchronousExecutionAndScaleOut, builds on the asynchronous execution sample and illustrates how to handle many common design issues and requirements that come with asynchronous processing. The sample shows how to perform the following tasks:

- Scale out to multiple processors by using batch tasks.
- Clean up the results table periodically.
- Detect errors in asynchronous operations.
- Use the alerts framework for notifications.



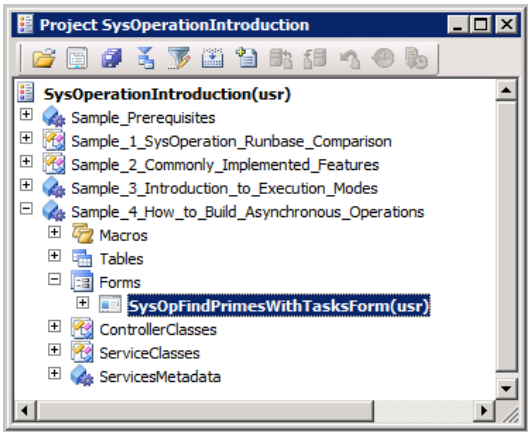
Sample overview

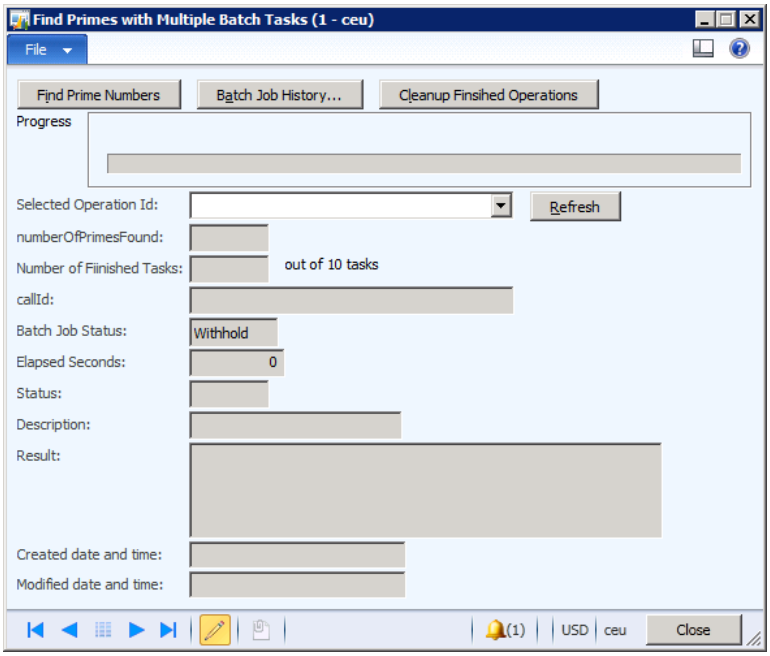
In this sample, the user interface is enhanced. It contains a progress bar that is updated as the operation runs and a **Result** text box that shows output from the operation logic. Note that, in this sample, the operation runs only via the batch server, so the **Batch Job Status** field from the BatchJobHistory table is prominent in the UI.

Scaling out to multiple processors by using batch tasks

This sequence shows the computation of prime numbers, as in the previous sample. The range in which primes are to be found is subdivided among multiple batch tasks.

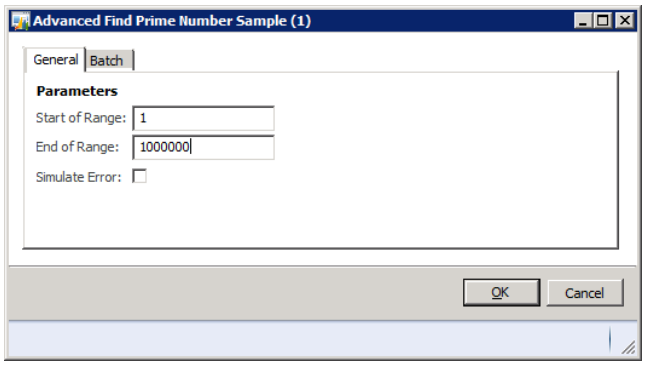
1. Open **SysOpFindPrimesWithTasksForm**.



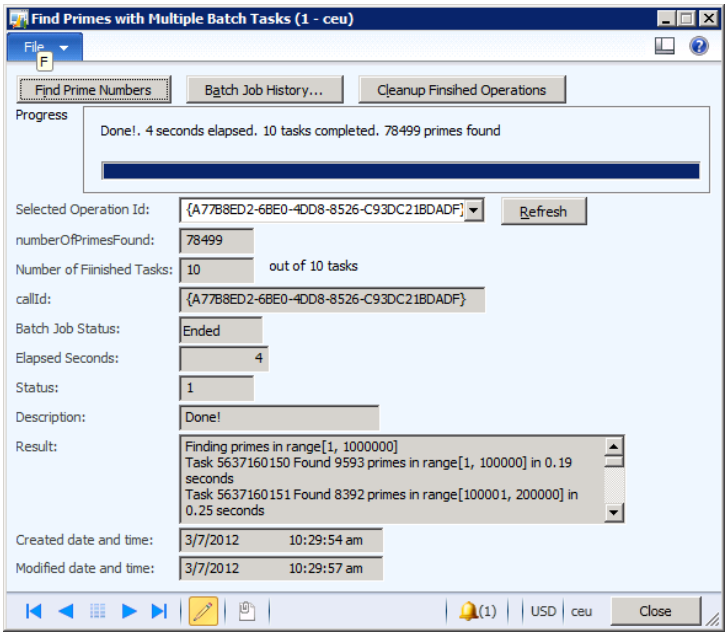


2. Click **Find Prime Numbers**, and then enter the operation input parameters. As in the previous sample, enter the range **[1,1000000]**.

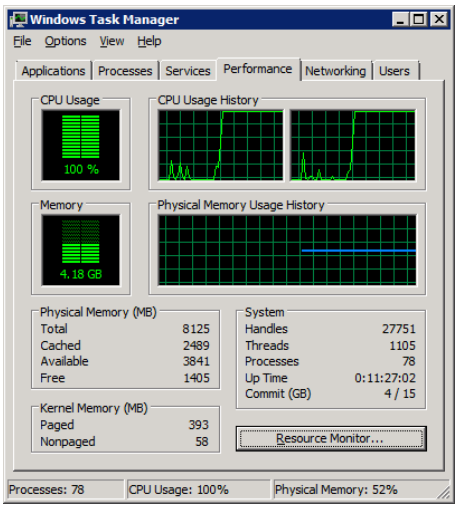
3. Make sure that the **Simulate Error** check box is cleared. Also make sure that the **Batch processing** check box is cleared on the **Batch** tab.



4. Click **OK** to run the operation.



If you have multiple processors in your machine, note that the full power of all the processors **is** used in the computation. For large ranges, compare the execution time with the previous sample. The time difference is not great for small ranges because of the overhead of coordinating multiple tasks.

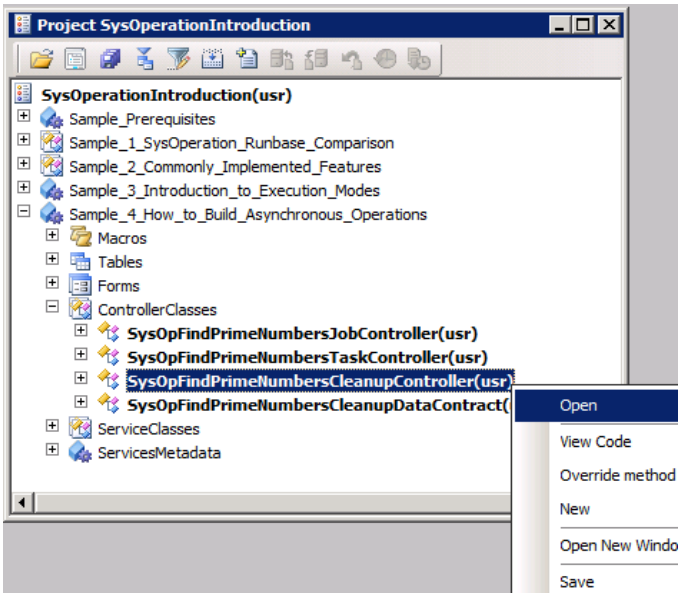


Cleaning up the results table periodically

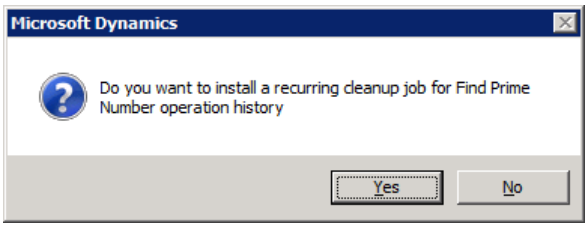
The design of this sample involves inserting a record in SysOpFindPrimesWithTasksTrackingTable for every operation. The record represents the status of the operation and also contains the results when the operation succeeds. Over time, as users run operations, the table will fill with records, so a mechanism is needed to periodically purge it. The controller class **SysOpFindPrimeNumbersCleanupController** is provided to periodically purge old operations.

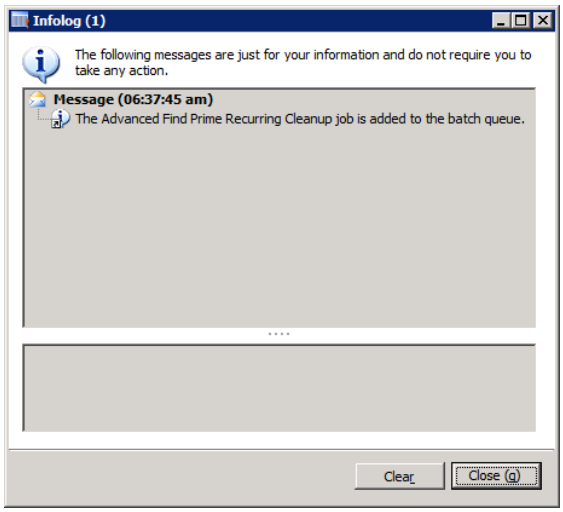
To install and run this controller:

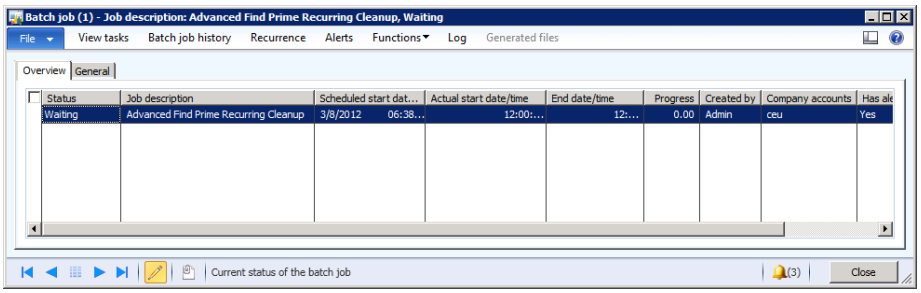
1. Right-click the controller class in the sample project, and open it.



2. Click **Yes** to install the controller as a recurring batch job.

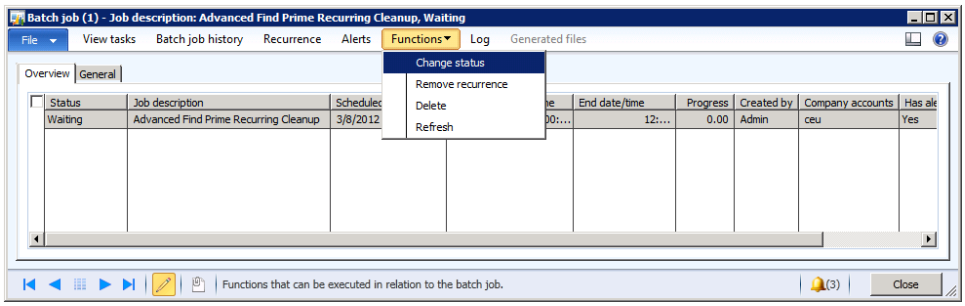


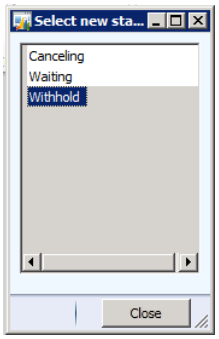




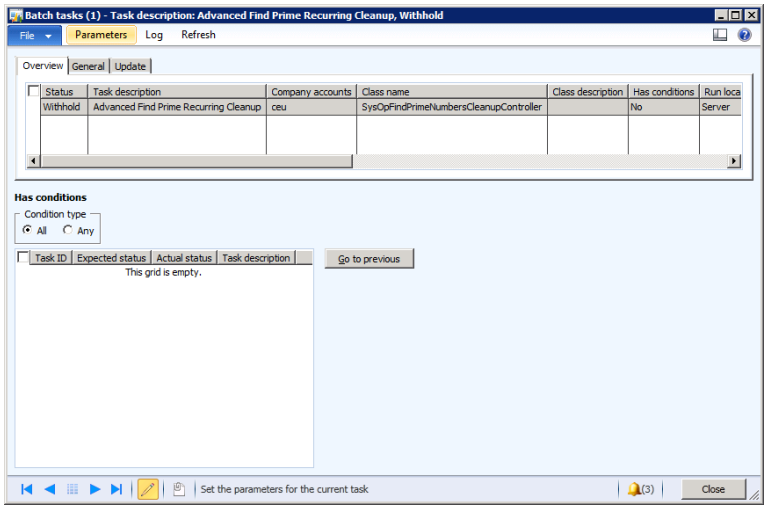
As configured, this batch job will clean up all operations that are more than 10 minutes old if they are not in an error state. Operations that were not completed successfully are left in the table, to alert the administrator of potential problems.

3. To configure the parameters for this recurring task, click **Change status** on the **Functions** menu in the **Batch job** form when the job is in the **Waiting** status. Change the status to **Withhold**.

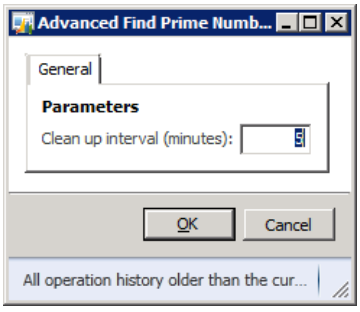




4. After the job is in the **Withhold** status, click **View tasks** on the toolbar in the **Batch job** form to open the **Batch tasks** form.

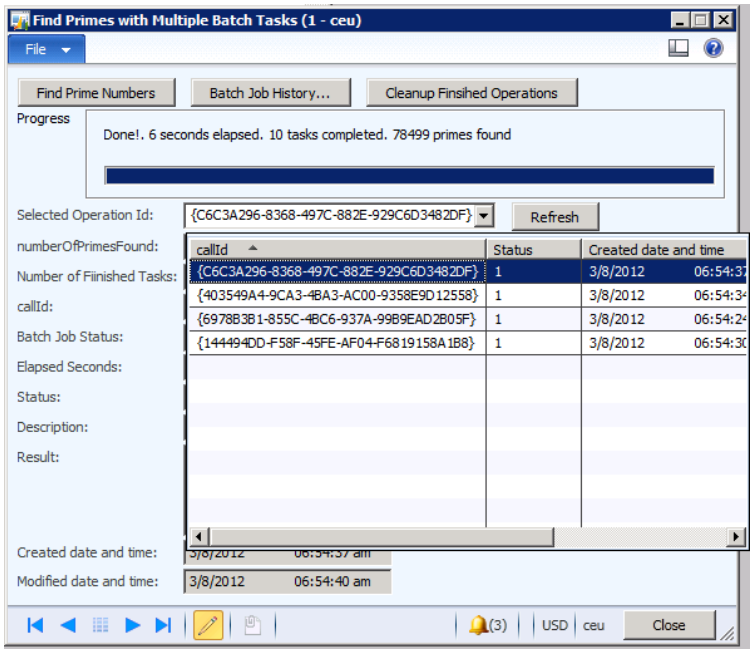


5. Click **Parameters** on the toolbar. This will instantiate the parameters dialog box for the cleanup operation. Note that this dialog box is provided by the SysOperation framework.

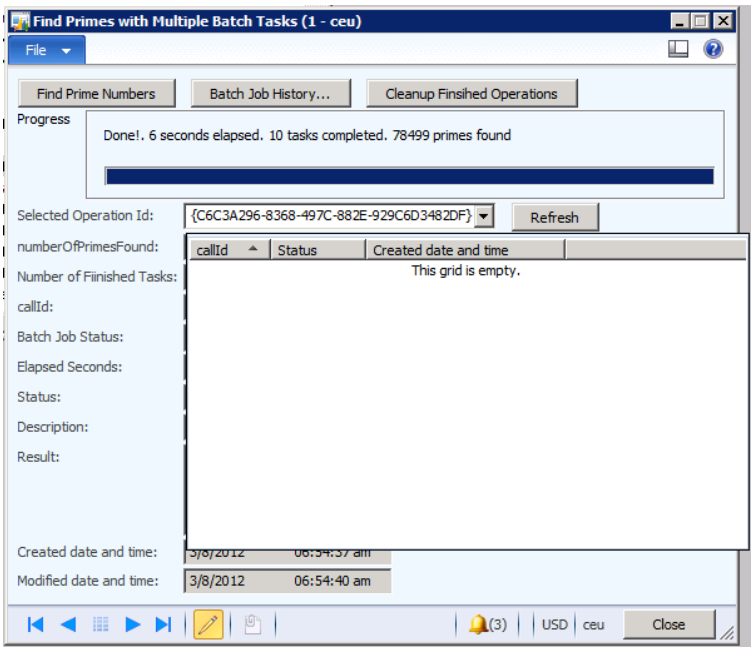


6. Modify the cleanup interval as desired, and then click **OK**. Close the **Batch tasks** form, and change the status of the parent batch job from **Withhold** to **Waiting**.
7. Run some operations.

8. Validate that all the operations exist in the tracking table by clicking the **Selected Operation Id** lookup.



9. Wait for the cleanup interval that you configured, and then click the **Selected Operation Id** lookup again to verify that the successful operations have been purged.

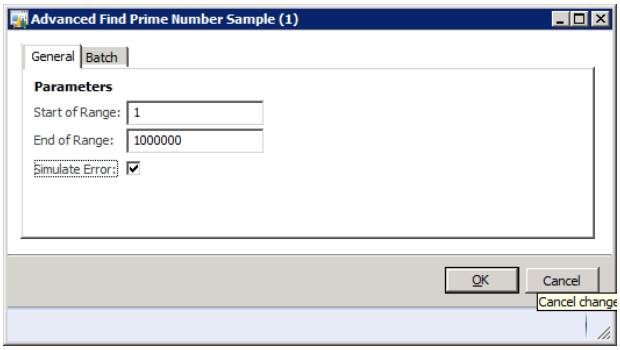


Detecting errors in asynchronous operations

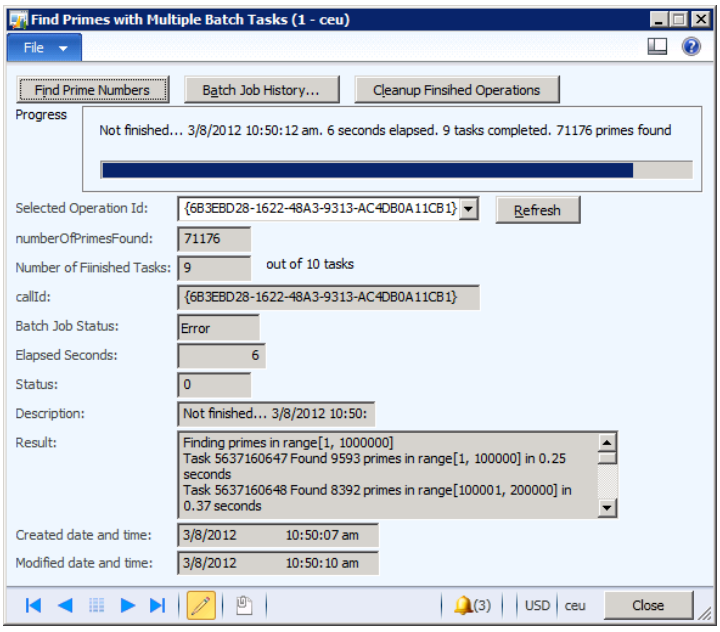
Error handling requires a little more thought in asynchronous scenarios. Errors happen in a different session, possibly in a different process and on a machine from where the caller is. The sample application uses the BatchJobHistory table to determine whether an error has occurred. The calling form polls that table to determine when the operation is completed. This is an indirect mechanism, so the sample also attempts to update the **Status** column in SysOpFindPrimesWithTasksTrackingTable to indicate that the task has been completed with an

error. This step cannot be done in the code executing the task, because an unhandled exception can cause the task code to exit. **SysOpFindPrimeNumbersCleanupController** is used to propagate the error information from the BatchJobHistory table to the tracking table. Make sure that the cleanup controller is installed by using the steps in the previous section, "Cleaning up the results table periodically", before completing the following steps.

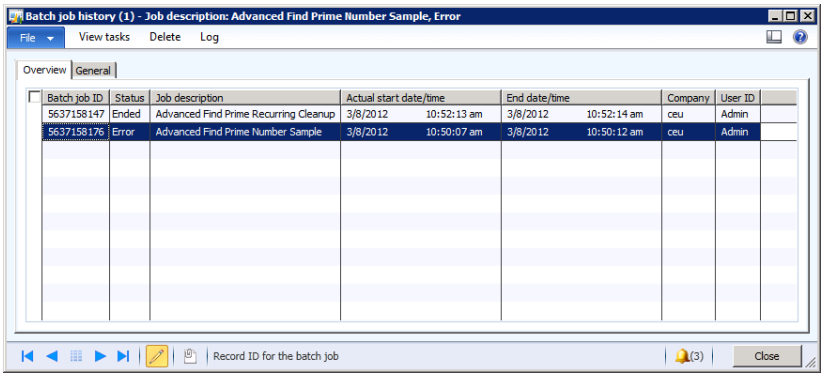
1. Run an operation, and select the **Simulate Error** check box in the operation input parameters form.



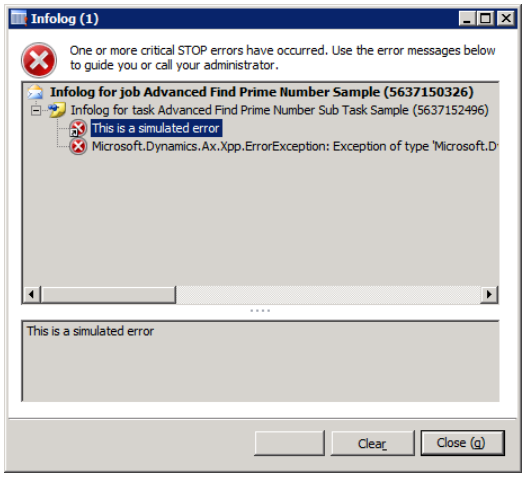
2. Wait for the operation to finish running. Notice that the **Batch Job Status** field shows an error.



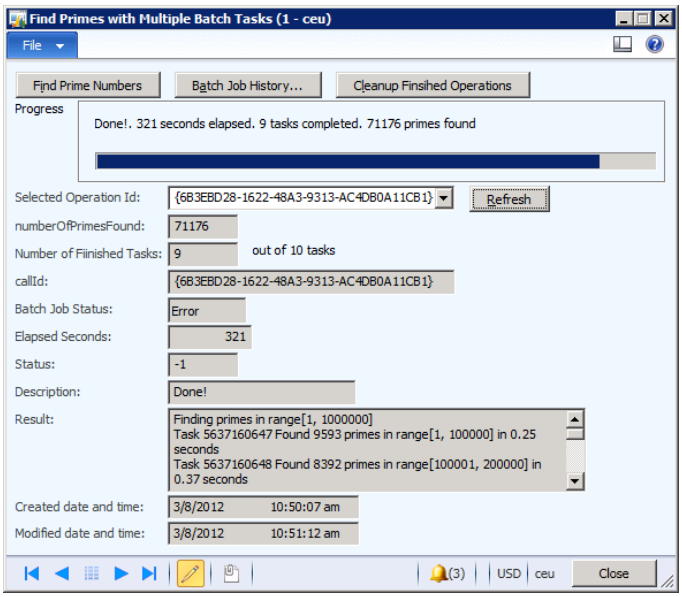
3. Click **Batch Job History** to open the **Batch job history** form.



4. Click **Log** to view the errors.



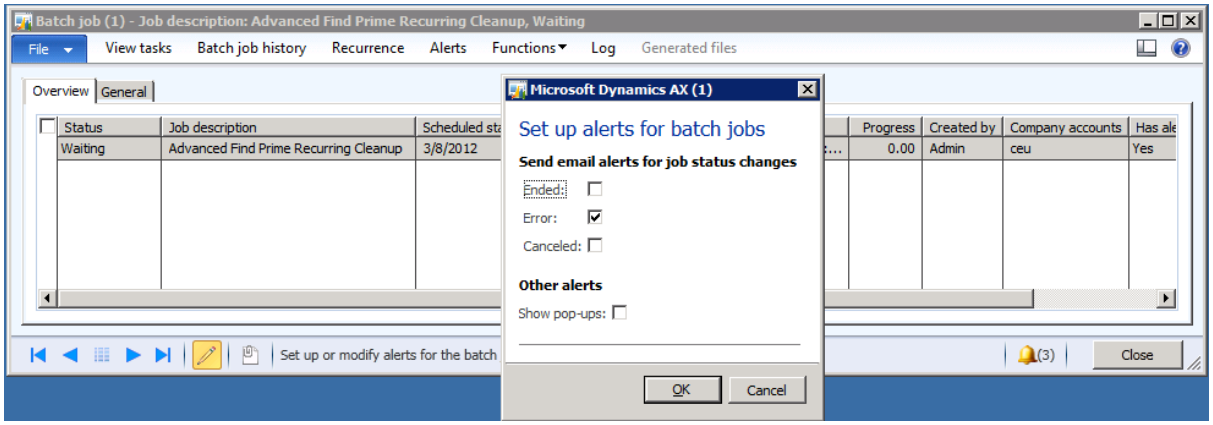
5. Wait a minute or two for the cleanup job to run. It is set to recur every minute. Click **Refresh** in the sample form. The **Status** field should be updated to **-1**, and the operation should be marked as **Done!** in the **Description** field.



The **-1** status on the operation record signifies that the operation did not end correctly. The caller can request more information as needed from the batch job history. Having the error status updated by a periodic job takes care of all exceptional scenarios. The task of catching all exceptions is assigned to the batch server. Application logic can handle the exceptions that it understands and is free to throw exceptions.

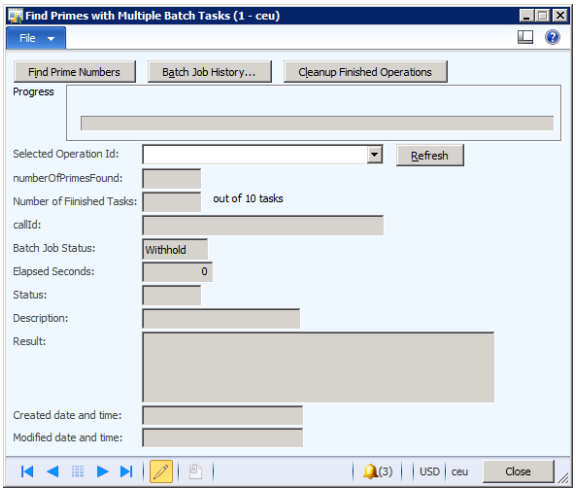
Using the alerts framework for notifications

Microsoft Dynamics AX provides an alerts subsystem that can be used to communicate operation status to administrators and end users. The batch server uses this subsystem to signal the status of batch jobs. Clicking the **Alerts** button in the **Batch job** form shows the current alert configuration for a given job.

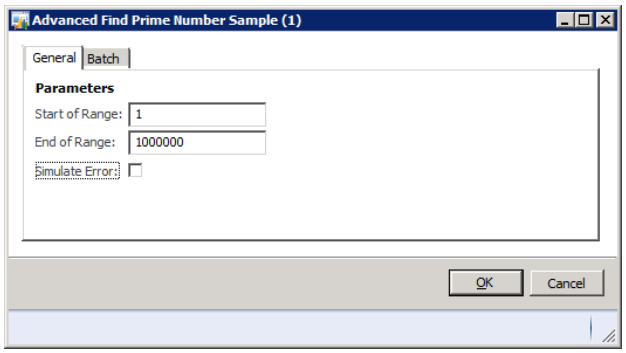


It is important to note that alerts are not generated for transient batch jobs created by the SysOperation **Reliable Asynchronous** execution mode. Transient batch jobs are batch jobs that have the property **parmRuntimeJob** set. Alerts can, however, be generated in application code as needed. To see alerts in action by using the default batch server functionality, use the form **SysOpFindPrimesWithTasksForm**.

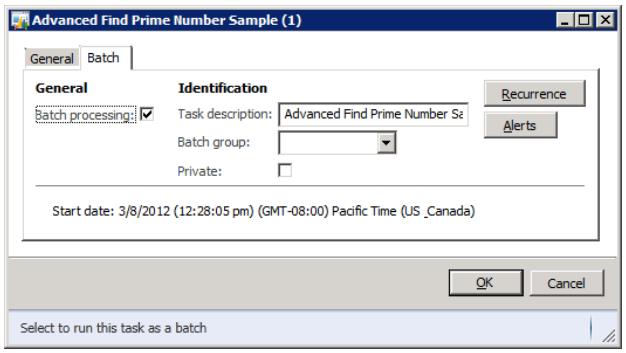
1. Open the form.



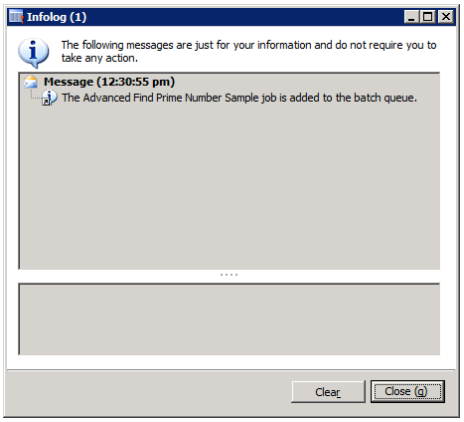
2. Click **Find Prime Numbers**, and then enter operation input parameters as before.



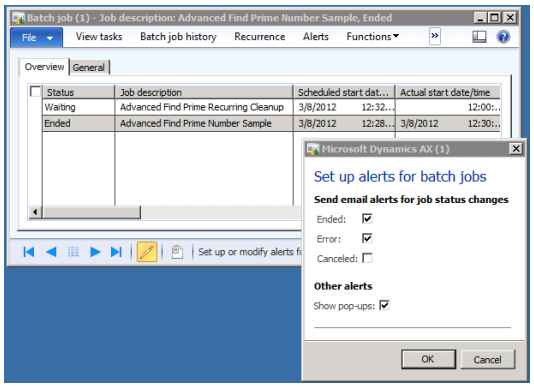
3. Select the **Batch processing** check box to run the operation via a non-transient job.



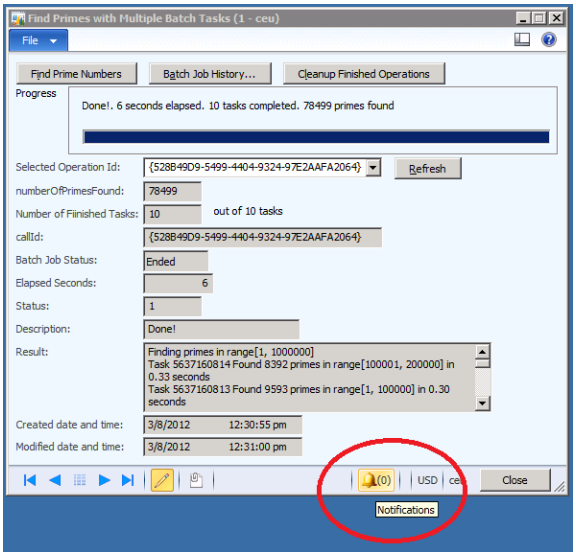
4. Click **OK** to run the job.

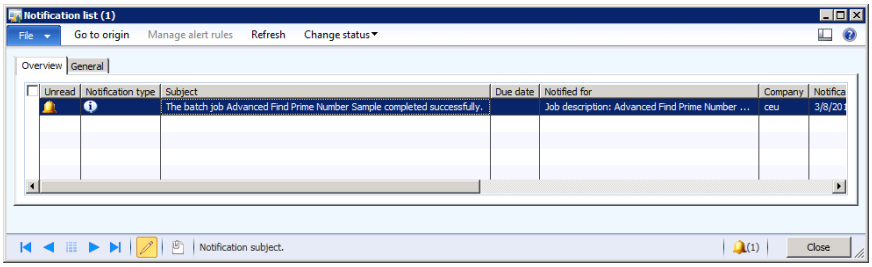


5. The job runs as before, but its entry persists in the batch job table. Navigate to the **Batch job** form to verify that the job is not automatically deleted upon completion. Look at its alerts configuration: it was configured in code and the settings differ from the default settings.



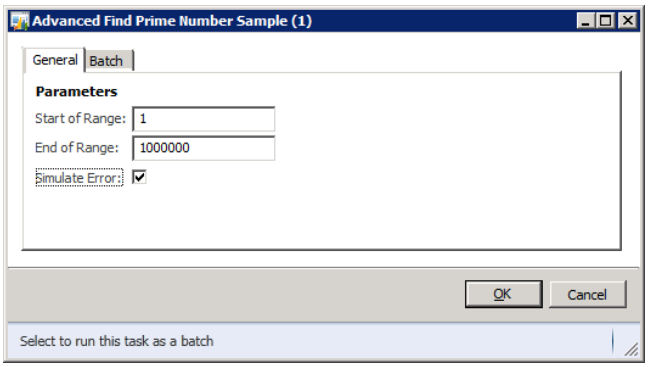
6. Click the **Notifications** (bell) button in the status bar at the bottom of the sample form. The notification count may not be updated immediately, but clicking the button will update it and open the **Notification list** form.

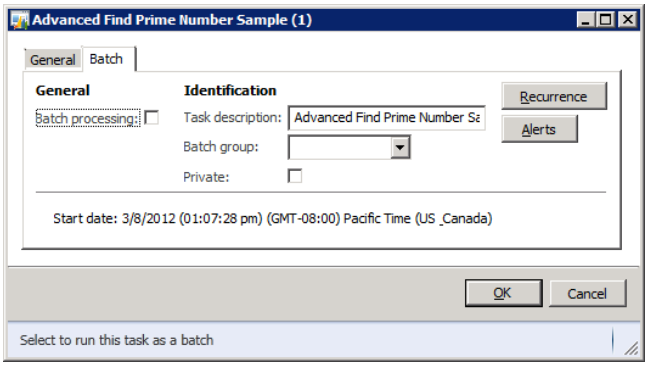




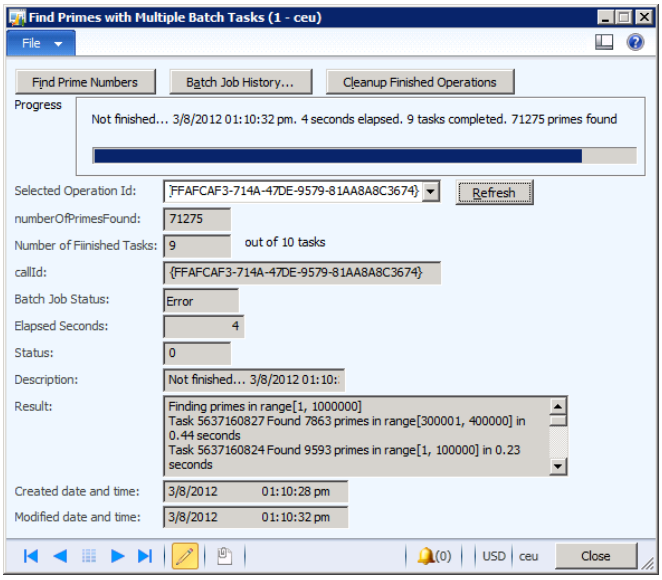
The standard batch server alerts may not be desirable for asynchronous operations where the use of the batch server is an implementation detail. The sample generates alerts in the asynchronous error propagation scenario. Make sure that the cleanup controller is installed by using the steps in the "Cleaning up the results table periodically" section before completing the following steps.

1. Run an operation with the **Simulated Error** check box selected and the **Batch processing** check box cleared.

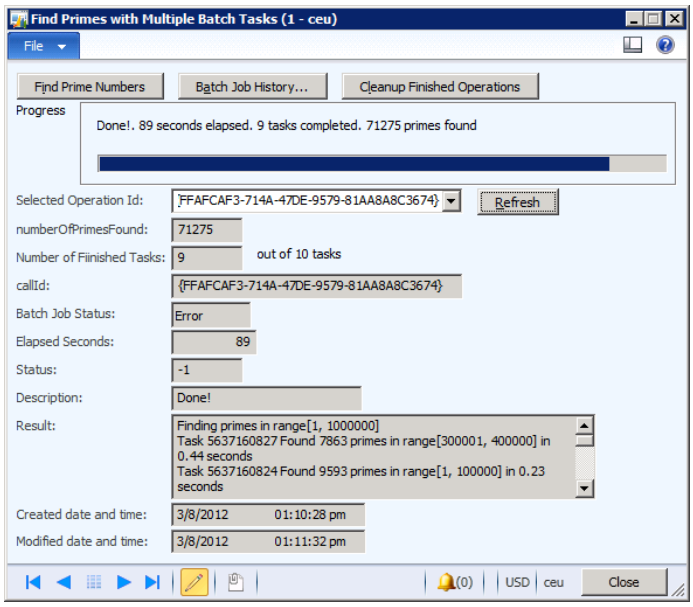




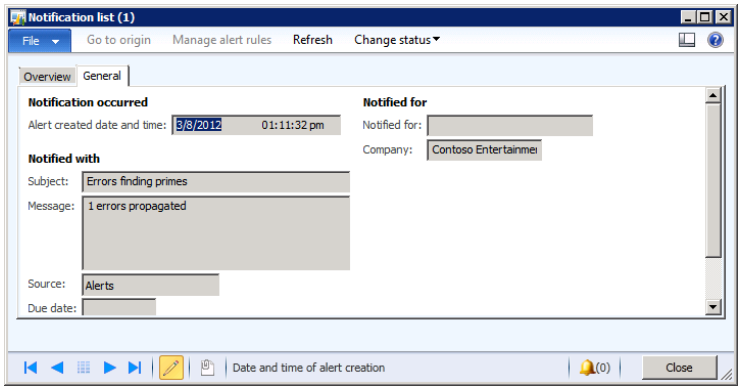
2. Click **OK** to run the operation in **Reliable Asynchronous** execution mode.



3. Wait a minute, and then click **Refresh** until the **Status** field shows **-1** and the **Description** field shows **Done!**

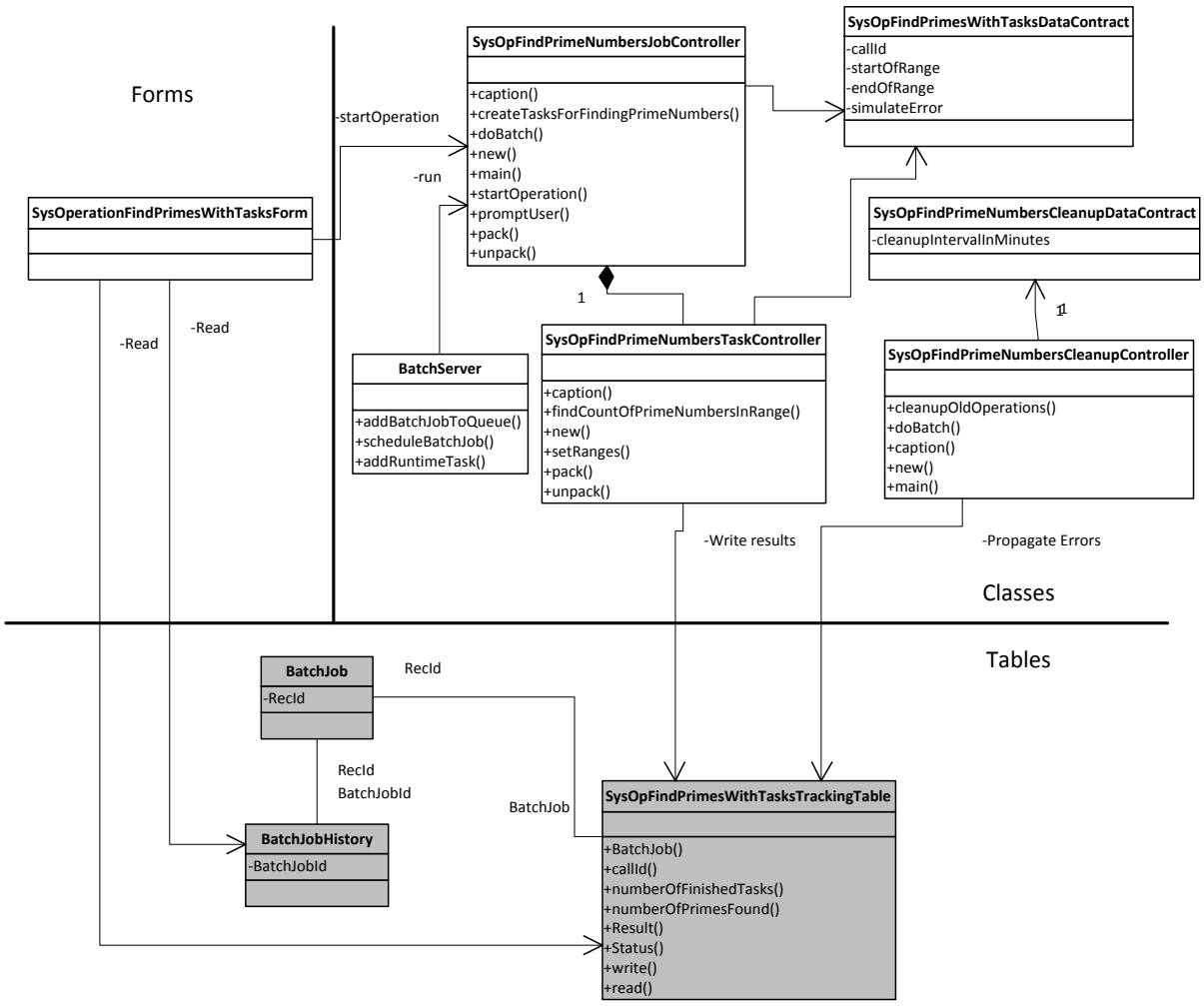


4. Click the **Notifications** (bell) button in the status bar at the bottom of the form. The notification count may not be updated immediately, but clicking the button will update it and open the **Notification list** form. A custom notification with the subject **Errors finding primes** should be in the list.

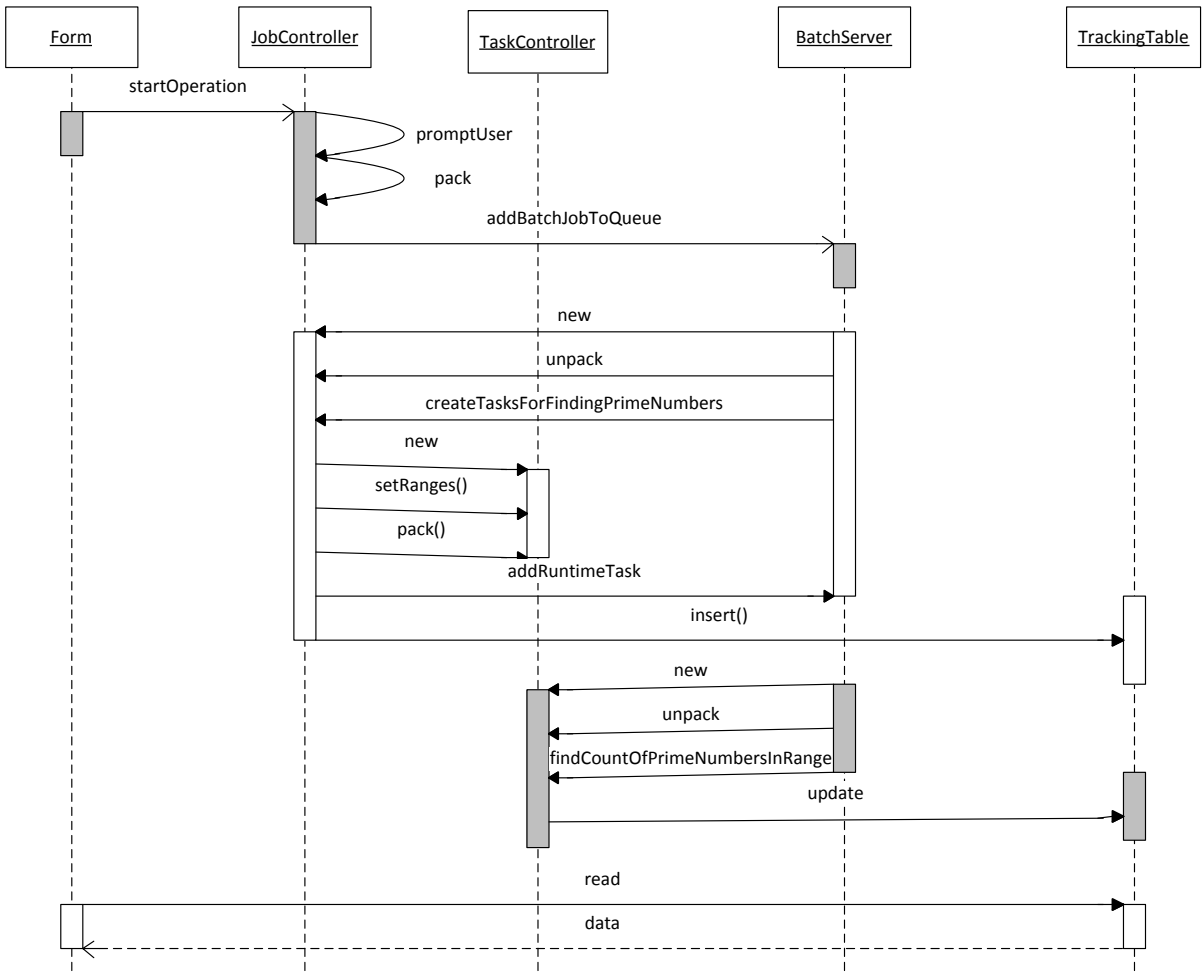


Architecture and code

Class diagram



Sequence diagram



Create multiple run-time tasks.

SysOpFindPrimeNumbersJobController

```
public void createTasksForFindingPrimeNumbers(SysOpFindPrimesWithTasksDataContract range)
{
    #SysOpFindPrimesWithTasks

    int64 i, start, end, blockSize, remainder;
    SysOpFindPrimesWithTasksTrackingTable logTable;
    RefRecId batchJobId;
    BatchHeader batchHeader;
    BatchJob currentJob;
    SysOpFindPrimesWithTasksDataContract subRange;
    SysOpFindPrimeNumbersTaskController taskController;

    // This sample can only function in batch mode
    if (!this.isInBatch())
    {
        throw AifFault::fault('SysOpFindPrimeNumbersJobController must run in batch mode',
            'NotInBatch');
    }
```

```
    // Get the header of the currently executing job
    batchHeader = BatchHeader::getCurrentBatchHeader();
    batchJobId = batchHeader.parmBatchHeaderId();

    // the delete on success flag (runtimeJob) is lost when altering batch header.
    // Restoring it. This parameter is set when the job is created to execute in reliable
    // async mode
    select RuntimeJob from currentJob where currentJob.RecId == batchJobId;
    batchHeader.parmRuntimeJob(currentJob.RuntimeJob);

    // break up to range to create tasks. The number of tasks is hardcoded in a macro
    start = range.parmStartOfRange();
    end = range.parmEndOfRange();

    blockSize = (end - start + 1) / #BatchTaskCount;
    remainder = (end - start + 1) - (blockSize * #BatchTaskCount);

    for (i = 0; i < #BatchTaskCount; i++)
    {
        end = start + blockSize;
        if (i == #BatchTaskCount - 1)
        {
            end += remainder;
        }

        // Create a controller for each sub task and add it
        // to the current job
        taskController = new SysOpFindPrimeNumbersTaskController();

        subRange = taskController.getDataContractObject();
        subRange.parmCallId(range.parmCallId());
        subRange.parmStartOfRange(start);
        subRange.parmEndOfRange(end - 1);
        subRange.parmSimulateError(range.parmSimulateError());

        batchHeader.addRuntimeTask(taskController, 0);
        start = end;
    }

    ttsBegin;
    // Save the current batch header along with all the newly
    // created tasks
    batchHeader.save();

    // Insert a record in the operation tracking table. All the tasks will
    // update this record.
    logTable.clear();
    logTable.BatchJob = batchJobId;
    logTable.Status = 0;
    logTable.callId = range.parmCallId();
    logTable.Result = strFmt('Finding primes in range[%1, %2]',
        range.parmStartOfRange(), range.parmEndOfRange());
    logTable.insert();
    ttsCommit;
}
```

Each task finds primes in its subrange.

SysOpFindPrimeNumbersTaskController

```
public void findCountOfPrimeNumbersInRange(SysOpFindPrimesWithTasksDataContract range)
{
    #SysOpFindPrimesWithTasks

    int64 i, primes;
    int ticks;
    SysOpFindPrimesWithTasksTrackingTable logTable;
    RefRecId batchJobId;

    // This controller can only run in batch mode
    if (this.isInBatch())
    {
        batchJobId = BatchHeader::getCurrentBatchHeader().parmBatchHeaderId();
    }

    // Compute prime numbers in the sub range
    ticks = WinAPIServer::getTickCount();
    primes = 0;
    for (i = range.parmStartOfRange(); i <= range.parmEndOfRange(); i++)
    {
        if (SysOpFindPrimeNumbersTaskController::isPrime(i))
        {
            primes++;
        }
    }

    ticks = WinAPIServer::getTickCount() - ticks;

    try
    {
        ttsBegin;

        // Hold a lock on the current row to avoid update conflicts with other threads
        // use forUpdate instead of pessimisticLock and you will see lots of retries
        // in the batch job history log
        select pessimisticLock logTable where logTable.callId == range.parmCallId();

        logTable.numberOfFinishedTasks++;
        // last thread updates the done status
        if (logTable.numberOfFinishedTasks == #BatchTaskCount)
        {
            // For simulated errors use the AIF fault mechanism. It allows application
            // errors to be differentiated from general unhandled errors
            if (range.parmSimulateError())
            {
                throw AifFault::fault('This is a simulated error', 'ApplicationError');
            }
            logTable.Status = 1;
        }

        logTable.numberOfPrimesFound += primes;
        logTable.callId = range.parmCallId();
        logTable.Result += strFmt('\r\nTask %4 Found %3 primes in range[%1, %2] in %5
seconds',
```

```
            range.parmStartOfRange(),
            range.parmEndOfRange(),
            primes,
            BatchHeader::getCurrentBatchTask().RecId,
            ticks / 1000);
        logTable.update();

        // Since all the tasks update the same record, there is significant
        // conflict on committing the transaction. There are better ways to do
        // this; each task could have its own tracking record
        ttsCommit;
    }
    catch (Exception::Deadlock)
    {
        retry;
    }
    catch (Exception::UpdateConflict)
    {
        // This will not be used with pessimistic locking
        // use ForUpdate instead of PessimisticLock in the select
        // to see this in action
        retry;
    }
}
```

Install a cleanup job programmatically.

SysOpFindPrimeNumbersCleanupController

```
public static void main(args a)
{
    BatchJob batchJob;
    SysOpFindPrimeNumbersCleanupController cleanupController;
    SysOpFindPrimeNumbersCleanupDataContract cleanUpParameters;
    batchInfo batchInfo;
    SysRecurrenceData recurrenceData;

    #define.JobName('Advanced Find Prime Recurring Cleanup');

    if (Box::yesNo('Do you want to install a recurring cleanup job for Find Prime Number
operation history', DialogButton::Yes) == DialogButton::Yes)
    {
        select RecId from batchJob where batchJob.Caption == #JobName;
        if (batchJob)
        {
            warning('Batch job for recurring cleanup job of Find Prime Number operation
history already exists.');
        }
        else
        {
            // instantiate the cleanup controller
            cleanupController = new SysOpFindPrimeNumbersCleanupController();

            // set its parameters
            cleanUpParameters = cleanupController.getDataContractObject();
            cleanUpParameters.parmCleanupIntervalInMinutes(10);

            // set up the recurrence information
            batchInfo = cleanupController.batchInfo();
```

```
            batchInfo.parmCaption(#JobName);
            recurrenceData = SysRecurrence::defaultRecurrence();
            // start in x minutes from now so that the job can be inspected via the batchjob
            // form before it starts.
            recurrenceData = SysRecurrence::setRecurrenceStartDateTime(recurrenceData,
                DateTimeUtil::addMinutes(DateTimeUtil::utcNow(), 1));
            recurrenceData = SysRecurrence::setRecurrenceNoEnd(recurrenceData);

            // Set the minimum recurrence interval of 1 minute
            recurrenceData = SysRecurrence::setRecurrenceUnit(recurrenceData,
                SysRecurrenceUnit::Minute, 1);
            batchInfo.parmRecurrenceData(recurrenceData);

            // This will add the job to the batch table
            cleanupController.parmExecutionMode(SysOperationExecutionMode::ScheduledBatch);
            cleanupController.doBatch();
        }
    }
}
```

Override the default batch job notification settings.

SysOpFindPrimeNumbersJobController

```
public Batch doBatch()
{
    BatchHeader batchHeader;

    if (executionMode == SysOperationExecutionMode::ScheduledBatch)
    {
        // Set up alerts so that they alert on success or error via a popup toast
        // If the job is set up for scheduled batch
        batchHeader = this.batchInfo().parmBatchHeader();
        batchHeader.clearAllAlerts();

        batchHeader.addUserAlerts(curUserId(),  // alert user who created the job
            NoYes::Yes,                         // completed
            NoYes::Yes,                         // error
            NoYes::No,                          // canceled
            NoYes::Yes,                         // popup or toast in desktop client
            NoYes::No);                         // email
    }
    else if (executionMode == SysOperationExecutionMode::ReliableAsynchronous)
    {
        // Alerts don't fire in reliable async mode so clear them
        batchHeader = this.batchInfo().parmBatchHeader();
        batchHeader.clearAllAlerts();

        batchHeader.addUserAlerts(curUserId(),  // alert user who created the job
            NoYes::No,                         // completed
            NoYes::No,                         // error
            NoYes::No,                          // canceled
            NoYes::No,                         // popup or toast in desktop client
            NoYes::No);                         // email
    }

    return super();
```

```
}
```

Create notifications from application code (and cleanup logic).

SysOpFindPrimeNumbersCleanupController

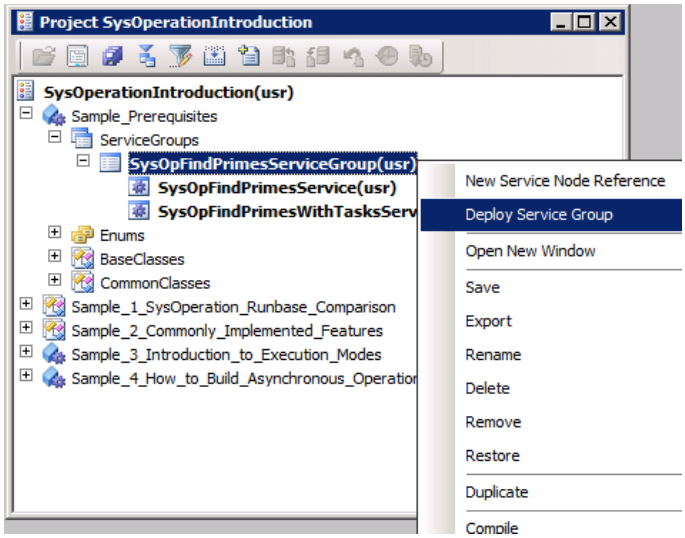
```
public void cleanupOldOperations(SysOpFindPrimeNumbersCleanupDataContract _input)
{
    SysOpFindPrimesWithTasksTrackingTable operationTable;
    BatchJobHistory jobHistory;
    utcDateTime minimumDateTime;
    int64 operationTableCount;
    int64 errorCount;
    int interval;
    EventNotificationBatch alerts;

    interval = _input.parmCleanupIntervalInMinutes();
    if (interval <= 0)
    {
        interval = 30;
    }

    // Clean up anything that is more than X minutes old where X is passed in
    // via the input parameter
    ttsBegin;
    minimumDateTime = DateTimeUtil::addMinutes(DateTimeUtil::utcNow(), -interval);

    // Get a count of records that will be cleaned up
    select count(RecId) from operationTable exists join jobHistory
            where
                jobHistory.Status == BatchStatus::Finished &&
                jobHistory.BatchJobId == operationTable.BatchJob &&
                operationTable.modifiedDateTime < minimumDateTime;

    operationTableCount = operationTable.RecId;

    // clean up the operation table. Don't delete errors
    delete_from operationTable exists join jobHistory
            where
                jobHistory.Status == BatchStatus::Finished &&
                jobHistory.BatchJobId == operationTable.BatchJob &&
                operationTable.modifiedDateTime < minimumDateTime;

    // Clean up the batch job history where there are no operation records
    delete_from jobHistory notexists join operationTable
        where
            jobHistory.BatchJobId == operationTable.BatchJob;

    ttsCommit;

    // bubble up error from batch history into operation table
    // set status to -1 where the batch job is in error
    ttsBegin;
    // get the count of errors that need to be propagated
    select count(RecId) from operationTable
        where
            operationTable.Status == 0
        exists join jobHistory
```

```
            where
                jobHistory.Status == BatchStatus::Error &&
                jobHistory.BatchJobId == operationTable.BatchJob;
    errorCount = operationTable.RecId;

    update_recordSet operationTable setting
        Status = -1
        where
            operationTable.Status == 0
        exists join jobHistory
            where
                jobHistory.Status == BatchStatus::Error &&
                jobHistory.BatchJobId == operationTable.BatchJob;

    ttsCommit;

    // Generate alert for erroneous operations
    if (errorCount)
    {
        ttsBegin;
        alerts = EventNotification::construct(EventNotificationSource::Batch);
        alerts.newInfo(curUserId(),                 // user
                       'Errors finding primes',     // subject
                       null,                        // menu item
                       null,                        // record
                       true,                        // popup
                       false,                       // email
                       '',                          // email address
                       curext(),                    // company
                       DateTimeUtil::utcNow(),      // date
                       strfmt('%1 errors propagated', errorCount)); // message
        alerts.create();
        alerts.insertDatabase();
        ttsCommit;
    }

    info(this.caption());
    info(strFmt('Cleaned up %1 operation records. %2 errors detected.', operationTableCount,
errorCount));
}
```

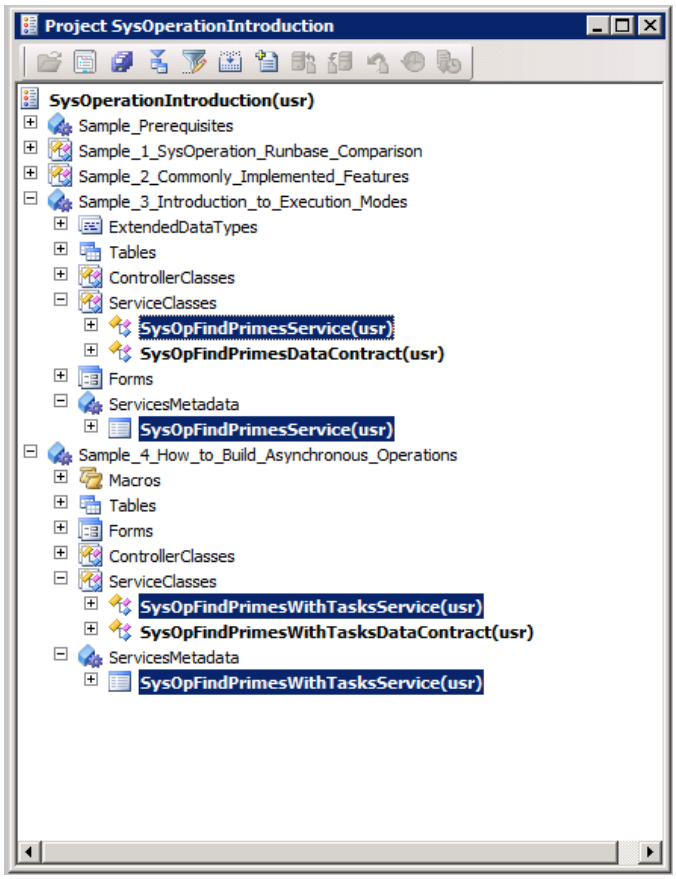
Note that, in this sample, the class **SysOpFindPrimesWithTasksService** is not used. It will be used in the next section, from the .NET client, to invoke the controller discussed here.

Sample 5: How to call asynchronous operations from .NET clients

So far, all the samples have used Microsoft Dynamics AX forms to drive the controllers built with the SysOperation framework. This sample will show how to use the same asynchronous execution patterns from a .NET client. The architecture is the same as for the Microsoft Dynamics AX forms. The .NET client runs a **Reliable Asynchronous** operation on the Microsoft Dynamics AX server, and then polls periodically for the operation results. This sample requires that the service group deployed at the beginning of this paper be available.



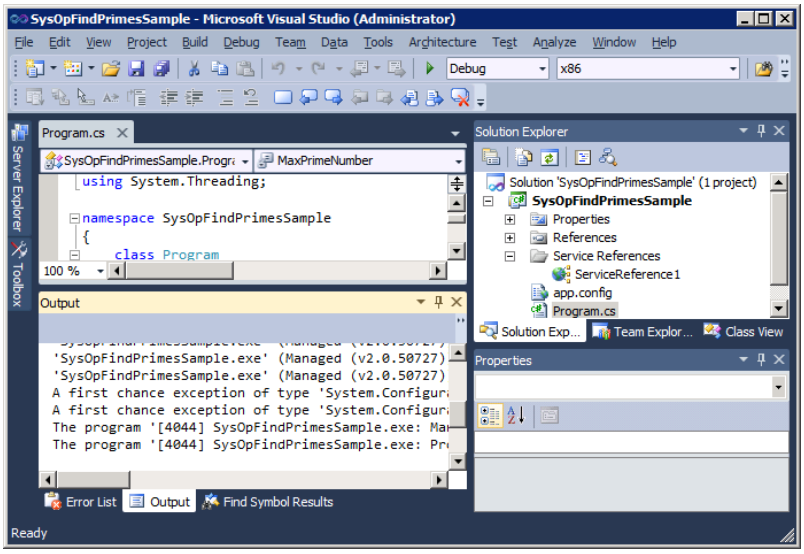
Deploying this group creates a WCF endpoint out of the Microsoft Dynamics AX server. The endpoint exposes two service interfaces. Four artifacts in the sample project define the service interfaces exposed by the endpoint.



The first service, SysOpFindPrimesService, runs the controller that computes primes by using a single batch task. The second service, SysOpFindPrimesWithTasks, computes primes by using multiple batch tasks. The .NET client calls both service interfaces in the common endpoint and provides a comparison between the two designs.

To run the .NET client:

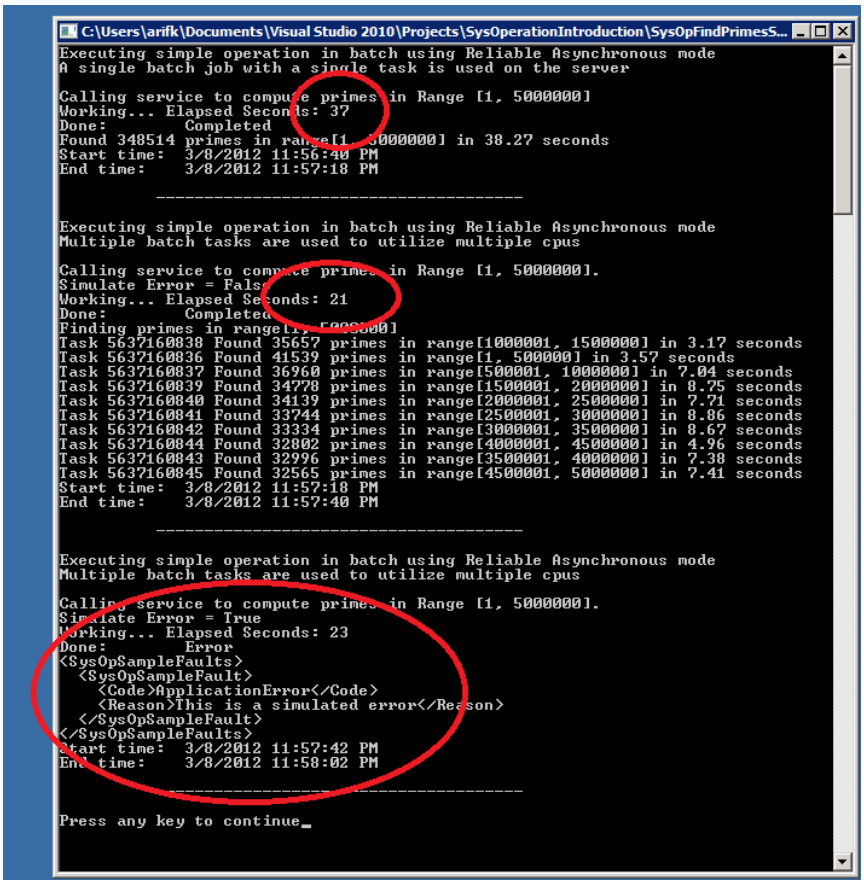
1. Open the solution **SysOpFindPrimesSample**.



2. Check the app.config file to make sure that the service endpoints point to **localhost**. It is assumed that the sample will be run on the same machine as the Microsoft Dynamics AX server.
3. Run the sample. It will run in three steps:
    1. Run with a single batch task
    2. Run with multiple batch tasks

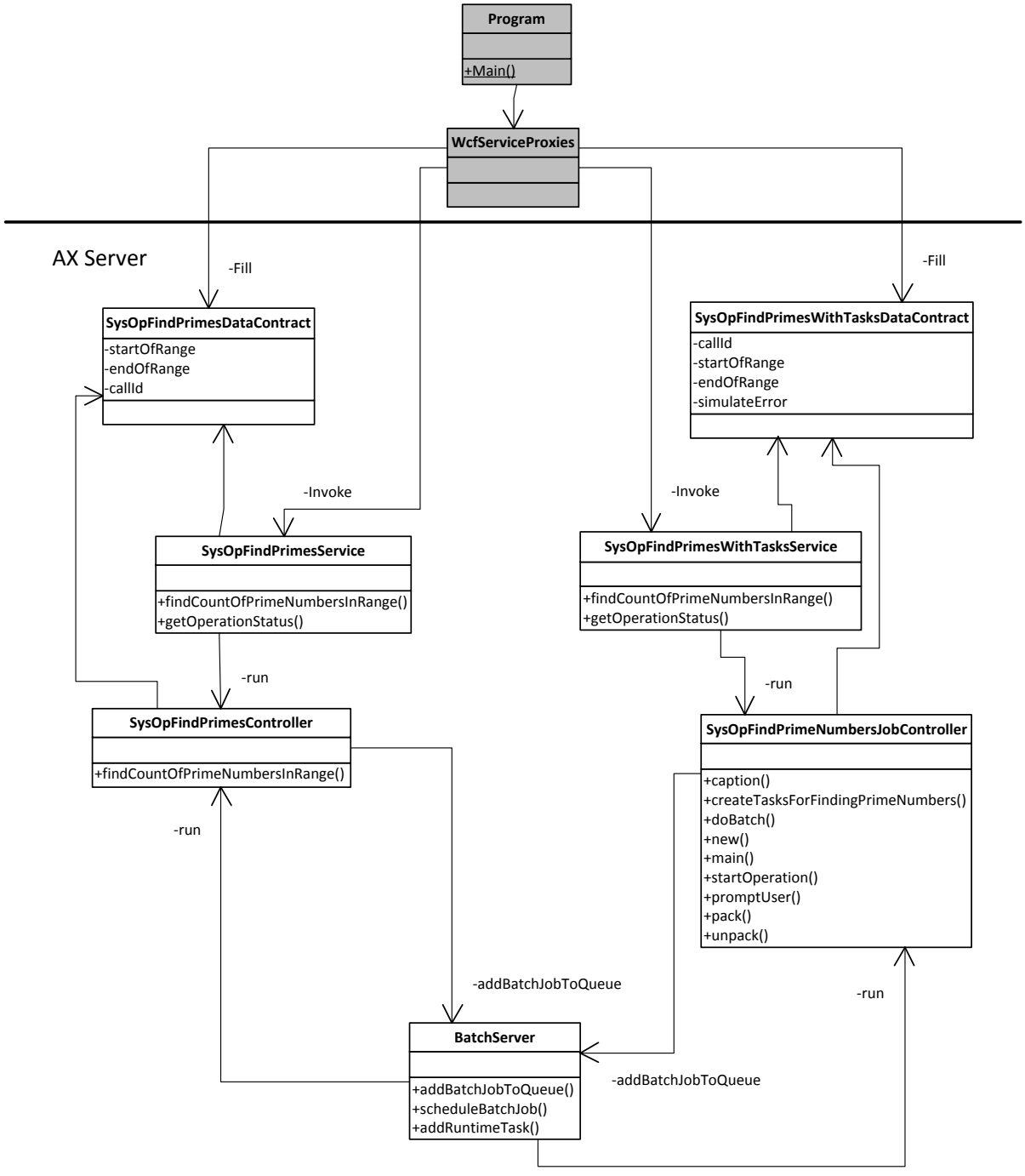
       This should run faster on a multiprocessor machine.
    3. Run with multiple batch tasks and simulate an error in one of the tasks

The error is propagated from the batch history to the .NET client.

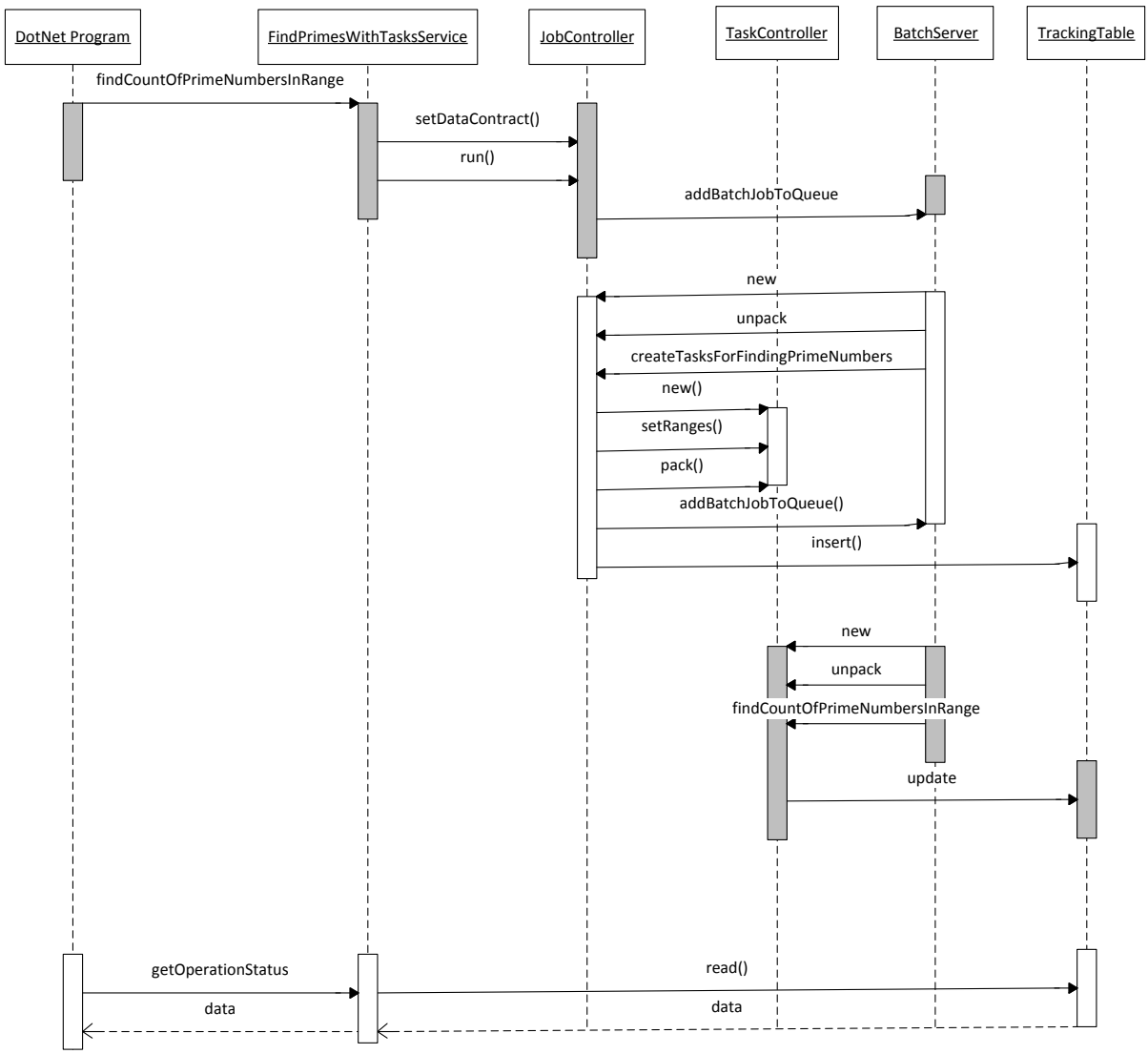


Architecture and code

Class diagram



Sequence diagram



Appendix: Workarounds for issues in the framework

There are multiple workarounds for framework issues in the **SysOpSampleBaseController** class. This class is used as a base for all controllers in the sample projects. The issues will be addressed in a future service pack.

Issue 1: The controller should not be unpacked from the SysLastValue table when running via batch.

Class SysOpSampleBaseController

```
protected void loadFromSysLastValue()
{
    if (!dataContractsInitialized)
    {
        // This is a bug in the SysOperationController class
        // never load from syslastvalue table when executing in batch
        // it is never a valid scenario
        if (!this.isInBatch())
```

```
        {
            super();
        }

        dataContractsInitialized = true;
    }
}
```

Issue 2: The default value for property parmRegisterCallbackForReliableAsyncCall should be false to avoid unnecessary polling of the batch server.

```
public void new()
{
    super();

    // defaulting parameters common to all scenarios

    // If using reliable async mechanism do not wait for the batch to
    // complete. This is better done at the application level since
    // the batch completion state transition is not predictable
    this.parmRegisterCallbackForReliableAsyncCall(false);

    … code removed for clarity …
}
```

Issue 3: The default value for property parmExecutionMode should be Synchronous to avoid issues when creating run-time tasks.

Class SysOpSampleBaseController

```
public void new()
{
    … code removed for clarity …

    // default for controllers in these samples is synchronous execution
    // batch execution will be explicitly specified. The default for
    // SysOperationServiceController is ReliableAsynchronous execution
    this.parmExecutionMode(SysOperationExecutionMode::Synchronous);
}
```

Issue 4: The value of the column runTimeJob in the BatchJob table is overwritten when runtime tasks are added to a batch job.

SysOpFindPrimeNumbersJobController

```
public void createTasksForFindingPrimeNumbers(SysOpFindPrimesWithTasksDataContract range)
{
    … code removed for clarity …

    // Get the header of the currently executing job
    batchHeader = BatchHeader::getCurrentBatchHeader();
    batchJobId = batchHeader.parmBatchHeaderId();

    // the delete on success flag (runtimeJob) is lost when altering batch header.
    // Restoring it. This parameter is set when the job is created to execute in reliable
    // async mode
```

```
    select RuntimeJob from currentJob where currentJob.RecId == batchJobId;
    batchHeader.parmRuntimeJob(currentJob.RuntimeJob);

    … code removed for clarity …
}
```

Updates since initial publication

The following table lists changes made to this document after it was initially published.

Date	Change
March 2012	Initial publication
March 2013	**Sample 3: Introduction to SysOperation execution modes** was added to provide descriptions of the execution modes for the SysOperation framework that are designed to provide different options for managing the single-threaded constraint that is associated with Microsoft Dynamics AX sessions.

Microsoft Dynamics is a line of integrated, adaptable business management solutions that enables you and your people to make business decisions with greater confidence. Microsoft Dynamics works like and with familiar Microsoft software, automating and streamlining financial, customer relationship and supply chain processes in a way that helps you drive business success.

U.S. and Canada Toll Free 1-888-477-7989

Worldwide +1-701-281-6500

www.microsoft.com/dynamics