An Introduction to Job Server In Sage SalesLogix v8.0

Sage SalesLogix White Paper

Documentation Comments	This documentation was developed by Sage SalesLogix User Assistance. For content revisions, questions, or comments, contact the Sage SalesLogix writers at saleslogix.techpubs@sage.com.
Copyright	
Version	Version 8.0 112812
Trademarks	
Disclaimer	Sage has thoroughly reviewed this paper. All statements, technical information, and recommendations in this paper and in any guides or related documents are believed reliable, but the accuracy and completeness thereof are not guaranteed or warranted, and they are not intended to be, nor should they be understood to be, representations or warranties concerning the products described. Sage assumes no responsibility or liability for errors or inaccuracies with respect to this publication or usage of information. Further, Sage reserves the right to make changes to the information described in this manual at any time without notice and without obligation to notify any person of such changes.

Table of Contents

An Introduction to Job Server In Sage SalesLogix v8.0

Purpose

The Sage Job Server enables you to schedule single and recurring tasks for immediate or delayed execution, either manually or dynamically at run time. It executes tasks out of process, freeing up worker threads and memory for client sessions. Good candidates for this are long running tasks and resource intensive tasks.

- **Long running tasks** – Offload long running tasks to the Job Server so they run asynchronously. When user sessions do not have to wait for long running processes to complete, you will see fewer request time outs.

- **Intensive tasks** - Eliminate competition for computing resources by offloading processing/memory intensive tasks to the Job Server. The Job Server runs in a different process from the Web Host and thus has a separate thread pool. It can even be located on a separate server.

The Sage Job Server is a shared service that can be consumed by any Sage SalesLogix adaptive client, including Web, Windows, and Mobile. In the base product, Job Server is required for roll over of incomplete activities and updating remaining days of contracts. Web Client features that use Job Server are updating opportunities and updating and deleting leads from the Task Pane.

What you need to know

This white paper is written for software developers. It assumes you have experience customizing Sage SalesLogix with .NET and SData. You will use .NET 4.0 to code job classes that you want to execute with the Job Server. You can make use of the same Sage SalesLogix ORM framework and services that are available to you when you create your Web Client customizations. You will use SData endpoints and SData client libraries to interact with the Job Server to do such things as schedule jobs and get their status.

Scheduling and thread pool management is handled for you by the core component, the open source Quartz.NET library. It is a port of the Quartz Java project.

Terminology

The following terms are key to understanding the Job Service:

- **Scheduler** - The component responsible for starting jobs, managing the worker thread pool, and persisting execution state. May refer to the entire Windows service application or the scheduler component it hosts.

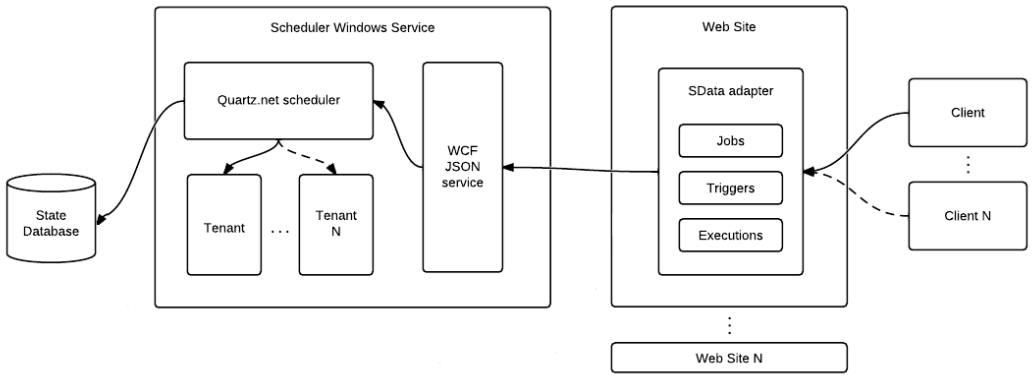
- **Tenant** - A folder containing the necessary assemblies and support files to execute a set of related jobs. Tenants are usually deployed from the Sage SalesLogix Job Service portal in Application Architect and can be identified by the existence of a tenant.config file in the folder root.
- **Job** - A class inheriting from SystemJobBase or UserJobBase that actually performs the work.
- **Trigger** - The mechanism by which a job is scheduled to run. Similar to an event, it causes a job to start executing. Often you will use an “immediate trigger” which executes a job on demand. A job can have multiple triggers. A single trigger may schedule one job run or recurring runs. Triggers that schedule recurring runs are called recurring triggers.
- **Execution** - An in-progress or recently completed job. Used to provide progress and result information. Recently completed executions are retained for 1 hour by default so you can collect information such as results and error messages. Recurring triggers result in multiple executions.

Architecture of the Sage Job Server

The following are the primary design features of the system:

- A scheduler that stores its state information in a database. All state information is stored in Sage SalesLogix tables prefixed with ‘QRTZ’.
- An API that uses standard JSON over HTTP, making it easy to call the service from a web site hosted in a DMZ. This API is exposed through a standard Sage SalesLogix application service, Sage.Scheduling.Client.ISchedulerService.
- An SData adapter that transparently handles the passing of user and tenant context. This is the normal way users consume the service. It provides a simple means of triggering jobs and listing all current and recent executions.
- Multi-tenancy support with AppDomain isolation and automatic binary update detection. When a fresh deployment occurs, the server suspends all triggers, interrupts running jobs, and reloads the updated tenant without the need for user intervention.
- A single scheduler can serve multiple webs sites. However, at this time, a single web site cannot be served by multiple schedulers with shared state.

Figure 1 Architecture of Sage Job Server



Record access security

If a job class inherits from UserJobBase, the job uses the security credentials of the user specified at run time (the logged-in user). For example, long running tasks initiated by end-users such as UpdateEntityJob and DeleteEntityJob typically run using the credentials of the initiating user. In contrast, if a job class inherits from SystemJobBase, the job uses the credentials of the Sage SalesLogix user associated to the Windows user the Job Server is logged on as. Night batch jobs such as UpdateRemainingDays (of Contract) fall into this category.

The Sage SalesLogix user needed by jobs inheriting from SystemJobBase should have access to all necessary data. Typically this is the ADMIN user or a service user that is a member of all teams.

Deployment

The Application Architect deployment infrastructure is used to deploy a file system, not an IIS portal. The file system deployment, Sage SalesLogix Job Service, is used for all Sage SalesLogix-related jobs.

The tenant folder, SlxJobService, is located under the base directory of the target which has the following default values:

C:\ProgramData\Sage\Scheduling\Tenants (for 2008/Win 7)

C:\Documents and Settings\All Users\Sage\Scheduling\Tenants (for XP/2003)

The deployment itself consists of a tenant.config file, a bin folder of static and platform built assemblies, and the usual deployment files such as application.xml, connection.config, and hibernate.xml.

Install location

The SData portal and the Sage Job Server service may reside on separate machines. The Sage Job Server service is commonly installed on the Sage SalesLogix application server where the “SalesLogix Server” service is installed.

Getting started

The following steps demonstrate the Job Server set up:

1. Install the Job Server according to *Sage SalesLogix Implementation Guide.pdf* or *Upgrading Sage SalesLogix from Version 7.5. to 8.0.PDF*. As part of the install or upgrade, you will have done the following:
    a. Install the Sage SalesLogix Job Service component as part of Admin Tools and Servers. The Sage Job Server service can be on a different server than the SData portal.

    b. Deploy the Sage SalesLogix Job Service file system and the SData portal. SlxJobService is included in the Core Portals deployment and both targets in the Remote Sales Client deployment. In some upgrade situations, you may have to add SlxJobService to these deployments. Refer to *Upgrading Sage SalesLogix from Version 7.5. to 8.0.PDF* for more information.

2. Verify that a Windows service called "Sage Job Service" is started.
   The service looks for files in subfolders of the Tenants folder. It loads the tenant.config file on start up.

3. Verify the service is running as a Windows user associated to a Sage SalesLogix user which gives the service access to the necessary entities. Typically, the SalesLogix user is the ADMIN user or a service user that is a member of all teams. Often, the Windows user is WebDLL. For more information on the log on user for the Sage Job Service, refer to *Sage SalesLogix Implementation Guide.pdf*.

4. Open the Sage.Scheduling.Server.exe.config file in a text editor and note the tenantRoot key value matches the base directory specified for the SageJobService deployment target in the Application Architect Deployment Explorer:

```
<add key="sage.scheduling.server.tenantRoot" value="%ALLUSERSPROFILE%\Sage\Scheduling\Tenants" />
```

5. Open the web.config of the SData portal in a text editor and confirm that the scheduling adapter is registered under the "Sage.Integration.Web.Adapters" element:

```
<Sage.Integration.Web.Adapters>
  <add type="Sage.Integration.Entity.Adapter.DynamicAdapter, Sage.Integration.Entity.Feeds"/>
  <add type="Sage.SalesLogix.SystemAdapter.SystemAdapter, Sage.SalesLogix.SystemAdapter"/>
  <add type="Sage.SalesLogix.ProxyAdapter.ProxyAdapter, Sage.SalesLogix.ProxyAdapter"/>
  <add type="Sage.Platform.SDataServices.MetadataAdapter, Sage.Platform.SDataServices"/>
  <add type="Sage.Platform.Mashups.Web.SData.MashupAdapter, Sage.Platform.Mashups.Web"/>
  <add type="Sage.Platform.Scheduling.SData.Adapter, Sage.Platform.Scheduling.SData"/>
</Sage.Integration.Web.Adapters>
```

6. Open the appSettings.config file of the SData portal in a text editor and confirm the following:

    a. The "sage.scheduling.client.serverUrl" setting is pointing to the correct host and port. For example:

```
<<add key="sage.scheduling.client.serverUrl" value="http://localhost:1895" />
```

b. The "sage.platform.scheduling.sdata.tenantId" setting matches the deployed portal folder. For example:

```
<add key="sage.platform.scheduling.sdata.tenantId" value="SlxJobService" />
```

7. Browse to the top level resource kind endpoint for scheduling and confirm that the three resource types appear. For example

   `http://<localhost:port>/sdata/$app/scheduling/-/?format=json`

   returns:

   - jobs
   - triggers
   - executions

8. Browse to the jobs endpoint to see a list of the jobs that can be scheduled to be executed. For example: `http://<machine:port>/sdata/$app/scheduling/-/jobs?format=json` You should get a list of jobs provided in the base product, such as Sage.SalesLogix.BusinessRules.Jobs.UpdateEntityJob.

   The SData adapter returns an error if the Job Service is not running or cannot be accessed due to network or authentication issues.

9. Browse to the triggers endpoint to see what jobs are scheduled to run.
   For example: http://<machine:port>/sdata/$app/scheduling/-/triggers?format=json

   The jobs in the base product that start out with triggers, such as Sage.SalesLogix.Activity.RolloverActivitiesJob, are already scheduled to execute. Note that repeatCount=-1 means the job is scheduled to repeat indefinitely. Jobs that do not have triggers are not scheduled yet. Some jobs are scheduled through the user interface. For example, clicking Update Opportunities or Update Leads in the Task Pane creates a trigger entity for the job, Sage.SalesLogix.BusinessRules.Jobs.UpdateEntityJob.

10. Demonstrate executing a job by updating a lead from the Web Client task pane:

    a. Select a lead in the Lead list view, and then click **Update Lead** in the task pane.

    b. Change the owner and click **OK**.
    A trigger is created to run Sage.SalesLogix.BusinessRules.Jobs.UpdateEntityJob immediately. The trigger is fired, the execution object is created, and finally it is modified to show the execution is finished.

    c. Press **F5** to see the change in the list view.

    d. Browse to the executions endpoint to see the information about the execution of the job. For example: http://<machine:port>/sdata/$app/scheduling/-/executions?format=json

In the result under $resources, note an entry with jobId= “Sage.SalesLogix.BusinessRules.Jobs.UpdateEntityJob” and status=”Complete”.

Creating a new job

To create a job that you can execute with the Sage Job Service, do the following:

1. Create a class to do the work you want accomplished:
   Inherit from UserJobBase or SystemJobBase in the Sage.Platform.Scheduling namespace and implement the OnExecute method. The ID of the job exposed in the API is the full type name of the class, for example, Sage.SalesLogix.Contract. UpdateRemainingDaysJob.

   The following example job declares a string parameter and immediately returns the parameter value when executed. All writable, simple type (strings, primitives, dates, enums and arrays of them), public properties are assumed to be input parameters.

```
public class EchoJob : SystemJobBase
{
    public string StringValue { get; set; }

    protected override void OnExecute()
    {
        Context.Result = StringValue;
    }
}
```

   See Appendix B for an example of a job that updates the remaining days of contracts and reports progress of the job. For more in depth examples of jobs, you can review the job implementations that are provided in the base product. They implement more advanced features such as deterministic (controlled) interruption, dependency injection, and more complex input parameter types. You can find one by first getting the assembly and class name of a job from the tenant.config file and then reflecting on the assembly in the bin folder at C:\ProgramData\Sage\Scheduling\Tenants\SlxJobService\bin. A good example to study is the FindDuplicatesJob class in the Sage.SalesLogix.Services.PotentialMatch assembly.

2. Copy your assembly to the SlxJobService\bin folder:
   (for 2008/Win 7) C:\ProgramData\Sage\Scheduling\Tenants\SlxJobService\bin
   (for XP/2003) C:\Documents and Settings\All Users\Sage\Scheduling\Tenants\SlxJobService\bin

3. Add the new job to the tenant.config file:

   a. Go to C:\ProgramData\Sage\Scheduling\Tenants\SlxJobService and open the tenant.config file.

   b. Add a node for the new job under the jobs section, and save the file.

It is not necessary to explicitly restart the server. All files including the tenant.config file are monitored for changes. When one or more changes are detected, all executing jobs are interrupted and the tenant is completely shut down and reinitialized.

4. At this point your job is ready to be scheduled for immediate or delayed execution.
5. Remember to copy the assembly and tenant.config files back into the Application Architect project.

Scheduling a job by creating triggers

After you have created the job class to do the work, you can schedule the job to run by creating a trigger entity for it. A trigger entity defines the name of the job, when it should start, and if it should be run multiple times. A trigger may be configured to start immediately or in the future. In the following sections, SData endpoints are used to demonstrate how to trigger, track, and interrupt jobs. See Appendix A for other ways to trigger jobs.

To follow along with the examples in the next sections, try a developer tool such as Poster that allows you to type in HTTP requests, set the body and content type, and inspect the results directly.

Triggering a job with SData immediately

The most common use of the scheduler is to immediately trigger a job and poll at regular intervals for execution progress. The following example shows how DeleteEntityJob can be executed.

Call the trigger service with a collection predicate and a JSON payload containing request parameters:

```
POST /sdata/$app/scheduling/-
        /jobs('Sage.SalesLogix.BusinessRules.Jobs.DeleteEntityJob')/$serviceB/trigger
        ?format=json

{
  "request": {
    "parameters": [
      {
        "name": "EntityName",
        "value": "Account"
      },
      {
        "name": "SelectedIds",
        "value": ["AA2EK0013023","AA2EK0013024","AA2EK0013025"]
      }
    ]
  }
}
```

Parameters can also be passed using the query string with underscore prefixes if you want to avoid building a JSON request payload:

```
POST /sdata/$app/scheduling/-/jobs('Sage.SalesLogix.BusinessRules.Jobs.DeleteEntityJob')/
```

```
$service/trigger?_EntityName=Account&_SelectedIds
=AA2EK0013023,AA2EK0013024,AA2EK0013025&format=json
```

The response payload contains a single "triggerId" value:

```
{
  "response": {
    "triggerId": "fb66f331-0a42-4209-a8b9-d4acbce0da69"
  }
}
```

Tracking execution

The server automatically creates an execution entity when a scheduled trigger fires. The execution entity contains information on the job and trigger, progress, elapsed time, status, and results. If you want to track execution progress or fetch an execution result, then use the executions endpoint with a triggerId expression predicate based on the value returned from the original service operation response:

```
GET /sdata/$app/scheduling/-/executions(triggerId eq 'fb66f331-0a42-4209-a8b9-
       d4acbce0da69')?format=json
```

The response contains a single execution entity:

```
{
  "$key": "NON_CLUSTERED6345333950377639 55",
  "jobId": "Sage.SalesLogix.BusinessRules.Jobs.DeleteEntityJob",
  "triggerId": "8009a75a-d0eb-4d45-849e-e7fc65367871",
  "scheduledFireTimeUtc": "\/Date(1317742738477)\/",Break914

  "fireTimeUtc": "\/Date(1317742738544)\/",
  "phase": null,
  "phaseDetail": null,
  "progress": 66.7,
  "elapsed": "00:00:1.7951322",
  "remaining": null,
  "status": "Running",
  "result": null,
  "state": [
    {
      "name": "EntityName",
      "value": "Account"
    },
    {
      "name": "SelectedIds",
      "value": ["AA2EK0013023","AA2EK0013024","AA2EK0013025"]
    }
  ]
}
```

An execution is complete when the status property contains the value "Complete". If the job returns a simple result, then this value can be found in the result property. To fetch complex results (such as full exception objects in the event an error occurred), the related result endpoint should be used. For example:

```
GET /sdata/$app/scheduling/-/executions(triggerId eq 'fb66f331-0a42-4209-a8b9-
       d4acbce0da69')/result?format=json
```

You should switch to using the execution ID (found in the $key property) after the first request because it is slightly more performant. The execution ID cannot be returned from the trigger service due to technical reasons outlined below.

The execution ID will not be available if the immediate trigger has been delayed due to heavy load or first time application context initialization.

Interrupting execution

Executions can be interrupted using the interrupt service operation:

```
POST /sdata/$app/scheduling/-/executions('NON_CLUSTERED6345333950377639 55')
       /$service/interrupt?format=json
```

Note that expression collection predicates are not supported when calling service operations. This means that at least one GET must be made on the executions endpoint to map the trigger ID to the associated execution ID before an interruption can be performed.

The interrupt service signals the job to stop prematurely. It is up to the job implementation to determine when it is safe to abort and whether internal state should be stored so that execution can be resumed at a future time.

Scheduling a recurring trigger

Triggers are persistent and can be added, edited, and deleted using the usual SData conventions. Triggers support simple recurrence based on a fixed number of occurrences at a fixed interval between two fixed dates.

```
POST /sdata/$app/scheduling/-/triggers?format=json
```

```
{
  "jobId": "Sage.SalesLogix.Services.PotentialMatch.DeDupJob",
  "startTimeUtc": "\/Date(1317741283000)\/",
  "endTimeUtc": "\/Date(1318338000000)\/",
  "repeatCount": 5,
  "repeatInterval": "1.00:00:00",
  "parameters": [
    {
      "name": "GroupId",
      "value": "p6UJ9A00024G"
    },
    {
      "name": "JobName",
      "value": "DeDupGroupJob.Account"
    }
  ]
}
```

The response payload contains the persisted trigger entity which now includes a couple of extra read-only properties:


```
{
  "$key": "04562cbb-139c-4108-b688-3702376cde27",
  "jobId": "Sage.SalesLogix.Services.PotentialMatch.DeDupJob",
  "startTimeUtc": "\/Date(1317741283000)\/",
  "endTimeUtc": "\/Date(1318338000000)\/",
  "repeatCount": 5,
  "repeatInterval": "1.00:00:00",
  "priority": 5,
  "status": "Normal",
  "timesTriggered": 0,
  "parameters": [
    {
      "name": "GroupId",
      "value": "p6UJ9A00024G"
    },
    {
      "name": "JobName",
      "value": "DeDupGroupJob.Account"
    }
  ]
}
```

Appendix A: Methods of Creating Triggers

There are multiple ways to create triggers:

- On start up of the Job Service through the tenant.config file
- Trigger via SData
  - JSON payload method
  - Query argument method
- Trigger via API
- Execute directly (in-process, synchronous)

Create triggers on startup

If you want a trigger to exist as soon as the Job Service starts, add one or more trigger nodes for the job in the tenant.config file.

Example tenant.config file

The following example configuration file demonstrates all configurable elements:

```
<?xml version="1.0"?>
<configuration>
  <configSections>
    <section name="scheduling"
             type="Sage.Scheduling.Configuration.SchedulingSection, Sage.Scheduling"/>
  </configSections>
  <scheduling xmlns="urn:Sage.Scheduling.Configuration">
<!--refer to Quartz documentation on plugins-->
    <plugins>
      <plugin type="MyNamespace.MyPlugin, MyAssembly" />
    </plugins>
<!--refer to Quartz documentation on listeners-->
    <listeners>
      <listener type="MyNamespace.MyListener, MyAssembly" />
    </listeners>
    <jobs>
<!--example of a job with no parameters -->
      <job type="MyNamespace.MyJob, MyAssembly" />
<!--example of a job with a parameter-->
      <job type="MyNamespace.MyJob_WithParameter, MyAssembly">
        <parameters>
          <parameter name="MyParam" value="MyValue" />
        </parameters>
      </job>
<!--example of a job with an indefinitely recurring trigger (repeatCount=-1) -->
      <job type="MyNamespace.MyJob_WithTrigger, MyAssembly">
        <triggers>
          <trigger id="EveryHour_Indefinitely" repeatInterval="01:00:00" repeatCount="-1" />
        </triggers>
      </job>
```

```
<!--example of a job with trigger with a parameter. Also shows optional repeatInterval and
priority. If two executions have the same fire time, the one with the higher priority gets access
to a worker thread first. -->
      <job type="MyNamespace.MyJob_WithTrigger_WithParameter, MyAssembly">
        <triggers>
          <trigger id="Weekly_TenTimes_Starting2012_HighPriority" repeatInterval="7.00:00:00"
                   repeatCount="10" startTimeUtc="2012-01-01T00:00:00" priority="10">
            <parameters>
              <parameter name="MyParam" value="MyValue" />
            </parameters>
          </trigger>
        </triggers>
      </job>
    </jobs>
  </scheduling>
</configuration>
```

Trigger via SData (JSON payload method)

This is the recommended way to start jobs on the client side. (Javascript or C# SData client libraries may be used.) Tenant and user context are handled automatically by the SData adapter, meaning the job is run in the correct Job Server tenant (deployment) using the current user's credentials.

```
var uri = new SDataUri("http://localhost/sdata/$app/scheduling/-")
    {
        CollectionType = "jobs",
        CollectionPredicate = "'Sage.SalesLogix.Jobs.UpdateEntitiesJob'",
        ServiceMethod = "trigger"
    };
var json = @"
  {
    request: {
      parameters: [
        {name: 'EntityTypeName', value: 'Sage.Entity.Interfaces.ILead'},
        {name: 'PropertyNames', value: ['Industry', 'Division']},
        {name: 'PropertyValues', value: ['_TEST_', '_SDATA-PAYLOAD_']},
        {name: 'SelectedIds', value: ['QDEMOA000FE6', 'QDEMOA000FE7']}
      ]
    }
  }";
var request = WebRequest.Create(uri.ToString());
request.Credentials = new NetworkCredential("admin", "");
request.Method = "POST";
request.ContentType = "application/json";
using (var writer = new StreamWriter(request.GetRequestStream()))
{
    writer.Write(json);
}
request.GetResponse();
```

Trigger via SData (query argument method)

This is an alternative to the SData example above except that parameters are passed in the query string rather than via a payload. Parameters passed this way must have their names prefixed with an underscore.

```
var uri = new SDataUri("http://localhost/sdata/$app/scheduling/-")
    {
        CollectionType = "jobs",
        CollectionPredicate = "'Sage.SalesLogix.Jobs.UpdateEntitiesJob'",
        ServiceMethod = "trigger",
        QueryArgs =
            {
                {"_EntityTypeName", "Sage.Entity.Interfaces.ILead"},
                {"_PropertyNames", "Industry,Division"},
                {"_PropertyValues", "_TEST_,_SDATA-URL_"},
                {"_SelectedIds", "QDEMOA000FE4,QDEMOA000FE5"}
            }
    };
var request = new SDataRequest(uri.ToString(), HttpMethod.Post, new AtomEntry())
    {
        UserName = "admin"
    };
request.GetResponse();
```

Trigger via WCF

The Job Server exposes a WCF endpoint that can be called directly on the server side using the ISchedulerService CAB service (this is what the SData adapter uses). Tenant and user context must be passed via the tenantId and AuthorizationToken parameters respectively.

```
var tenantId = ConfigurationManager.AppSettings["sage.platform.scheduling.sdata.tenantId"];
var scheduler = ApplicationContext.Current.Services.Get<ISchedulerService>(true);
var authProvider = ApplicationContext.Current.Services.Get<IAuthenticationProvider>(true);
scheduler.TriggerJob(
    tenantId,
    "Sage.SalesLogix.Jobs.UpdateEntitiesJob",
    new Dictionary<string, object>
    {
        {"AuthenticationToken", authProvider.AuthenticationToken},
        {"EntityTypeName", "Sage.Entity.Interfaces.ILead"},
        {"PropertyNames", new[] {"Industry", "Division"}},
        {"PropertyValues", new[] {"_TEST_", "_WCF_"}},
        {"SelectedIds", new[] {"QDEMOA000FE2", "QDEMOA000FE3"}}
    });
```

Execute directly (in-process, synchronous)

Job implementations can be instantiated and executed synchronously in-process. However, with the exception of unit testing, this is generally not recommended. As with the previous example, the user's application token must be specified since jobs are designed to be run outside of a stateful, authenticated ASP.net session.

```
var authProvider = ApplicationContext.Current.Services.Get<IAuthenticationProvider>(true);
var job = new UpdateEntitiesJob
    {
        AuthenticationToken = authProvider.AuthenticationToken.ToString(),
        EntityTypeName = "Sage.Entity.Interfaces.ILead",
        PropertyNames = new[] {"Industry", "Division"},
        PropertyValues = new[] {"_TEST_", "_IN-PROCESS_"},
        SelectedIds = new[] {"QDEMOA000FE0", "QDEMOA000FE1"}
    };
using (new SessionScopeWrapper())
{
    job.Execute(null);
}
```

Appendix B: Sample job

Sage.SalesLogix.Contract.UpdateRemainingDaysJob()

- This is a Job of type SystemJobBase (Sage.Platform.Scheduling.SystemJobBase) as opposed to UserJobBase (Sage.Platform.Scheduling.UserJobBase).
- This job is run on a fixed schedule defined in the tenant.config file.

This code demonstrates using the following job processing feedback properties:

- Phase
- PhaseDetail
- Progress
- Context.Result

```
// VS project references:
//   NHibernate
//   Quartz
//   Sage.Entity.Interfaces
//   Sage.Platform
//   Sage.Scheduling
//   System

using System;
using Quartz;
using Sage.Entity.Interfaces;
using Sage.Platform.Orm;
using Sage.Platform.Scheduling;
using Sage.SalesLogix.BusinessRules;
using Sage.SalesLogix.BusinessRules.Localization;
using Sage.SalesLogix.BusinessRules.Properties;

namespace Sage.SalesLogix.Contract
{
    /// <summary>
    /// Update the remaining (days) count for all <see cref="IContract"/> of type Days.
    /// </summary>
    [DisallowConcurrentExecution]
    [SRDescription(SR.Contract_UpdateRemainingDaysJob_Description)]
    internal class UpdateRemainingDaysJob : SystemJobBase
    {
        private static readonly string _entityDisplayName =
            typeof (IContract).GetDisplayName();

        protected override void OnExecute()
        {
            Phase = Resources.Job_Phase_Initialization;
            Log.Info(Phase);
            PhaseDetail = Resources.Job_Phase_Initialization_Initializing;
```

```
Log.Info(PhaseDetail);

using (var session = new SessionScopeWrapper())
{
    PhaseDetail = string.Format(Resources.Job_Phase_Initialization_ComposingQuery,
                                _entityDisplayName);
    Log.Info(PhaseDetail);

    // Note: Contract.TypeCode stores a PickListId rather than a Code or a Value
    var picklistItemId = BusinessRuleHelper.GetPickListItemIdByCode("Contract Type",
                                                                    "Days");

    PhaseDetail = string.Format(Resources.Job_Phase_Initialization_Retrieving,
                                _entityDisplayName);
    Log.Info(PhaseDetail);

    // Contracts that end in the future, OR contracts that already ended in the past
    // but do not have a value of zero
    var contracts = session.QueryOver<IContract>()
        .Where(c => c.TypeCode == picklistItemId &&
                    (c.EndingDate > DateTime.UtcNow ||
                     c.Remaining == null ||
                     c.Remaining != 0))
        .List<IContract>();

    if (contracts != null)
    {
        Phase = Resources.Job_Phase_ProcessingRecords;
        Log.Info(Phase);
        PhaseDetail = string.Format(
            Resources.Job_Phase_ProcessingRecords_TotalRecordsToProcess,
            _entityDisplayName, contracts.Count);
        Log.Info(PhaseDetail);

        var counter = 0;
        foreach (var contract in contracts)
        {
            Log.Info("Processing ContractId: " + contract.Id);
            contract.UpdateRemainingDays();
            contract.Save();

            // halt processing if interrupt requested by job server
            if (Interrupted)
            {
                PhaseDetail = Resources.Job_Phase_ProcessingRecords_InteruptRequested;
                Log.Info(PhaseDetail);
                return;
            }

            // update job progress percentage
            Progress = 100M*++counter/contracts.Count;
            Log.Info("Job progress set to: " + Progress);
        }
    }
    else
    {
        // no records to process
        PhaseDetail = string.Format(Resources.Job_Phase_Initialization_NoQualifying,
                                    _entityDisplayName);
```

```
                    Log.Info(PhaseDetail);
                }
            }

            Phase = Resources.Job_Phase_Finalization;
            Log.Info(Phase);
            PhaseDetail = Resources.Job_Phase_Finalization_CleanupAndCompletion;
            Log.Info(PhaseDetail);
        }
    }
}
```

Appendix C: WCF Service Summary

The following WCF operations report describes the available endpoints and the HTTP methods they support.

Uri	Method	Description
executions	GET	Service at http://localhost:1895/executions
executions/{executionId}	GET	Service at http://localhost:1895/executions/{EXECUTIONID}
	DELETE	Service at http://localhost:1895/executions/{EXECUTIONID}
executions/{executionId}/interrupt	POST	Service at http://localhost:1895/executions/{EXECUTIONID}/interrupt
executions/{executionId}/result	GET	Service at http://localhost:1895/executions/{EXECUTIONID}/result
jobs	GET	Service at http://localhost:1895/jobs
jobs/{tenantId}	GET	Service at http://localhost:1895/jobs/{TENANTID}
jobs/{tenantId}/{jobId}	GET	Service at http://localhost:1895/jobs/{TENANTID}/{JOBID}
jobs/{tenantId}/{jobId}/executions	GET	Service at http://localhost:1895/jobs/{TENANTID}/{JOBID}/executions
jobs/{tenantId}/{jobId}/interrupt	POST	Service at http://localhost:1895/jobs/{TENANTID}/{JOBID}/interrupt
jobs/{tenantId}/{jobId}/pause	POST	Service at http://localhost:1895/jobs/{TENANTID}/{JOBID}/pause
jobs/{tenantId}/{jobId}/resume	POST	Service at http://localhost:1895/jobs/{TENANTID}/{JOBID}/resume
jobs/{tenantId}/{jobId}/trigger	POST	Service at http://localhost:1895/jobs/{TENANTID}/{JOBID}/trigger
jobs/{tenantId}/{jobId}/triggers	GET	Service at http://localhost:1895/jobs/{TENANTID}/{JOBID}/triggers
triggers	GET	Service at http://localhost:1895/triggers
	POST	Service at http://localhost:1895/triggers
triggers/{tenantId}	GET	Service at http://localhost:1895/triggers/{TENANTID}
triggers/{tenantId}/{triggerId}	GET	Service at http://localhost:1895/triggers/{TENANTID}/{TRIGGERID}
	PUT	Service at http://localhost:1895/triggers/{TENANTID}/{TRIGGERID}
	DELETE	Service at http://localhost:1895/triggers/{TENANTID}/{TRIGGERID}
triggers/{tenantId}/{triggerId}/pause	POST	Service at http://localhost:1895/triggers/{TENANTID}/{TRIGGERID}/pause
triggers/{tenantId}/{triggerId}/resume	POST	Service at http://localhost:1895/triggers/{TENANTID}/{TRIGGERID}/resume