APPENDIX

Sample Workflow Project

The sample project provided in the appendix is designed as a review of many of the concepts discussed in this book. It also demonstrates how the techniques you've learned can be combined to build a full-featured application. Instead of giving you step-by-step instructions for building the solution, the complete project is available for you to download from `http://www.apress.com`. As you'll see, this project is fairly extensive and this will save you quite a bit of time (and typing).

Project Overview

The application provides a web page in which end users can enter requests, comments, or questions. Based on the topic selected, the request is placed in one of several queues for individuals to view and respond to. A second web page is provided to show the contents of these queues. Once a queue is selected, items are presented to be worked. Each queue can be configured in one of two modes: either the oldest request is automatically presented to be worked or all requests are displayed for the user to select one.

After a request has been worked, it might require a QC review step based on rules defined in a `Policy` object. (The implementation is very similar to one you created in Chapter 20.) Requests can also be rerouted to a different queue if necessary. A tracking extension is used to record the various events (started, assigned, reviewed, rerouted, and so on) so you can see how a particular request made its way through the workflow.

All the workflow functionality is provided through a web service. The web site uses the native .Net membership services, which gives the site the capability to "log in." The operator information is provided to the workflow so it can track which users worked on each request. Finally, the generic workflow features and queue logic is provided by a set of workflow activities and extensions that are implemented in a separate library project. This allows you to reuse this code in other applications.

I'll explain the solution in more detail later, but first, let's run the application so you can see what it does. You'll need to download the `Appendix.zip` file and extract this to an appropriate location.

Configuring the Database

The AppendixData folder contains the scripts you'll need to set up the database schema. First, create a SQL Server database for this solution. Expand the Create Scripts folder and run the included scripts in the following order:

- SqlWorkflowInstanceStoreSchema.sql
- SqlWorkflowInstanceStoreLogic.sql
- Config.sql
- Request.sql
- Tracking.sql

The scripts used to create the membership tables and procedures assume that there is a database named aspnetdb on the server that the connection string is referencing. This is the default database that the ASP.Net services use. If you do not already have this setup, create a database called aspnetdb and then run the InstallCommon.sql and InstallMembership.sql scripts (in that order).

■ **Caution** If you need to modify the database connection for your environment, make sure that you make the change to the configuration files; there are two places. The web.config file in the root folder of the web site project defines the connection string for the aspnetdb database used by the .Net services. There is also a web.config file in the root folder of the ServiceLayer project. This is used to configure the workflow persistence store as well as the application data that your custom activities will use.

Running the Application

Once your database has been configured, from Visual Studio, press F5 to start the application. The initial page should look like the one shown in Figure A-1.

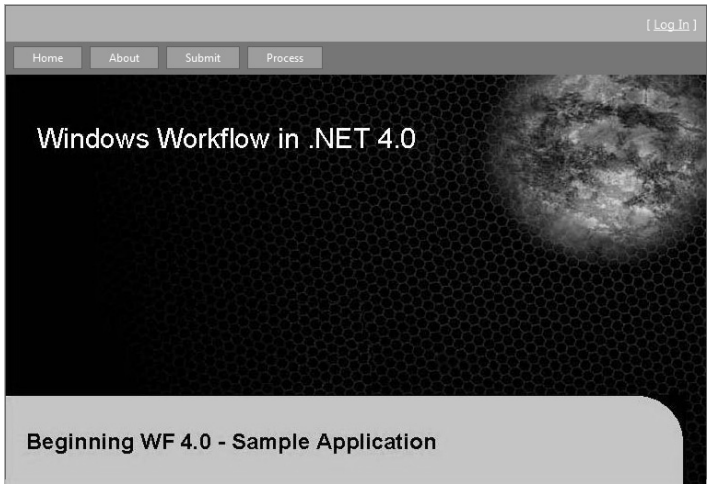


Figure A-1. Initial web page

Logging In

Click the Log In link at the top-right corner of the page. The first time you log in, you'll need to create a new account by clicking the register link. This will display the page shown in Figure A-2.

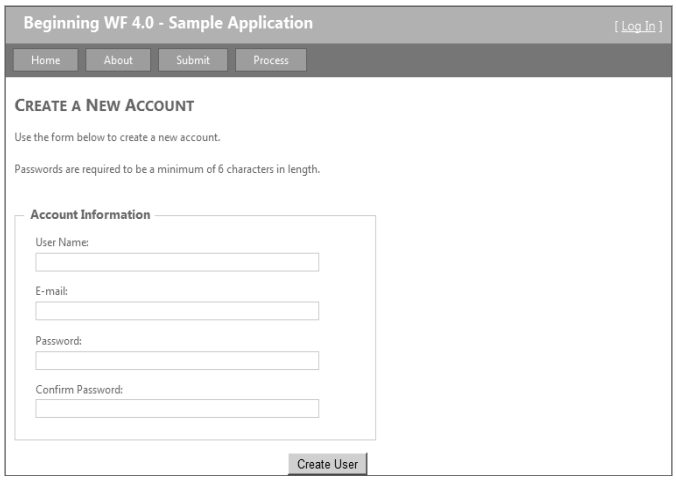


Figure A-2. Creating a new account

Enter your name, e-mail address, and a password (the e-mail address is not actually used to send e-mails so you can enter any text here that you want.) The next time you log in, you'll want to check the "Keep me logged in" check box, as shown in Figure A-3. If you do, you won't need to log in again even if you restart the application.

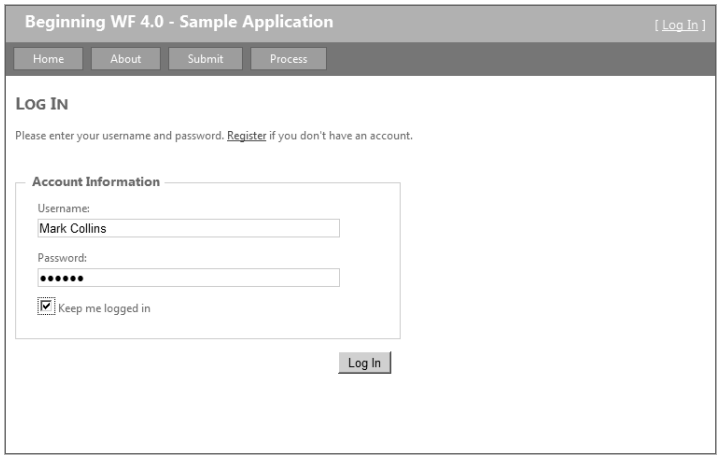


Figure A-3. Using the "Keep me logged in" option

Submitting a Request

Click the Submit link at the top of the page. This will display the page used for entering a new request/comment, which is shown in Figure A-4.

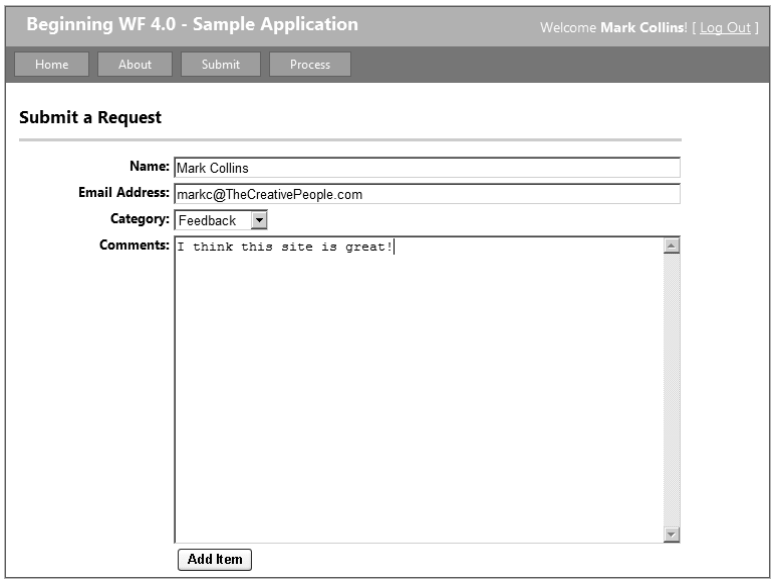


Figure A-4. Entering a new request

If you've logged in, the name and e-mail address will be filled in for you. Select the Feedback category and enter a comment. Then click the Add Item link. Figure A-5 shows a completed submit page. The comment field is cleared and the unique identifier assigned to this request is displayed at the top of the page. Enter a couple more requests using the same Feedback category.

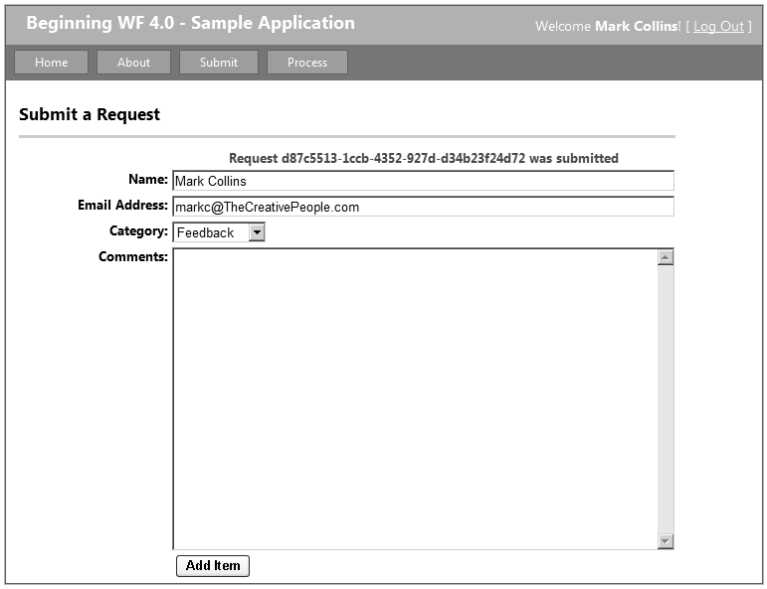


Figure A-5. The completed submit page

Processing Requests

Click the Process link at the top of the page. This will display the Process page that is shown in Figure A-6.

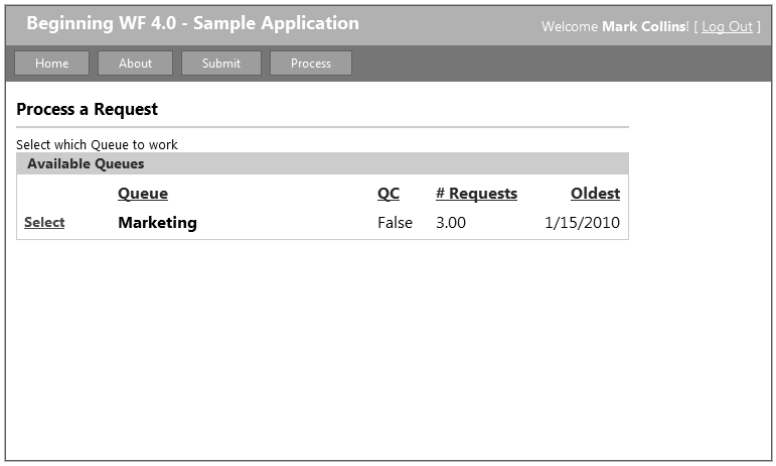


Figure A-6. Displaying the available queues

The grid on this page lists the queues that have requests that need to be responded to. The Feedback category was assigned to the Marketing queue so all the requests you entered are in this queue. This grid indicates how many requests are in this queue and the date of the oldest request. Click the Select link and the page should look like the one shown in Figure A-7.

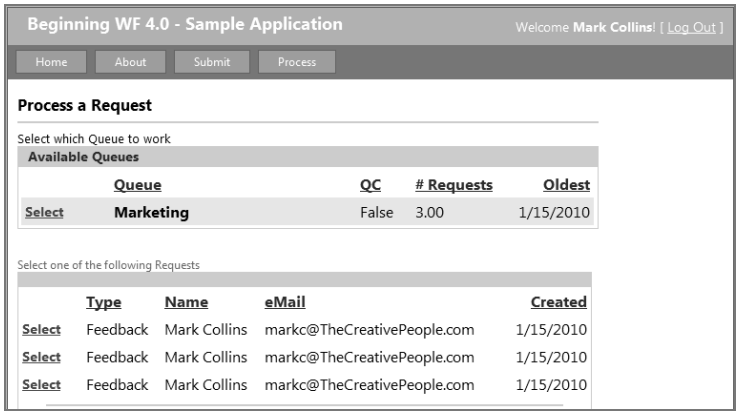


Figure A-7. Selecting a request

The Marketing queue is configured to allow selection. This means that instead of automatically assigning the oldest request in the queue, all the requests are listed, and you are allowed to select the one you want to work on. Click one of the Select links and the request will be displayed, as shown in Figure A-8.

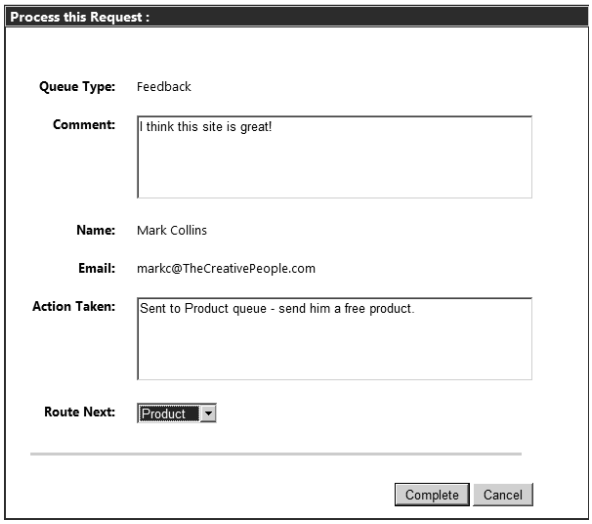


Figure A-8. Responding to a request

Below the request there is a place to enter the action that was taken. You can also select a queue to route this request to. You could click the Cancel button if you decided not to work on this request. That will unassign the request and put it back in the queue for someone else to work. Enter some notes in the Action Taken field, select the Product queue, and click the Complete button. The page should now look like the one shown in Figure A-9.

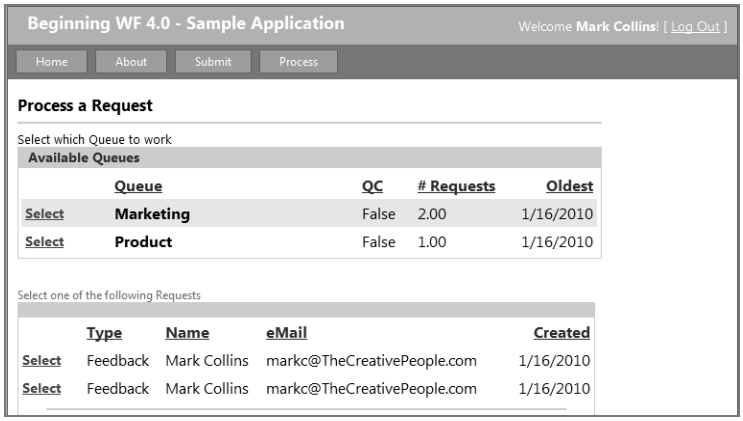


Figure A-9. The updated queue list

Notice that the Product queue is now displayed in the queue list, and there are only two requests to choose from in the Marketing queue. Click the Select link next to the Product queue. The page should look like the one shown in Figure A-10.

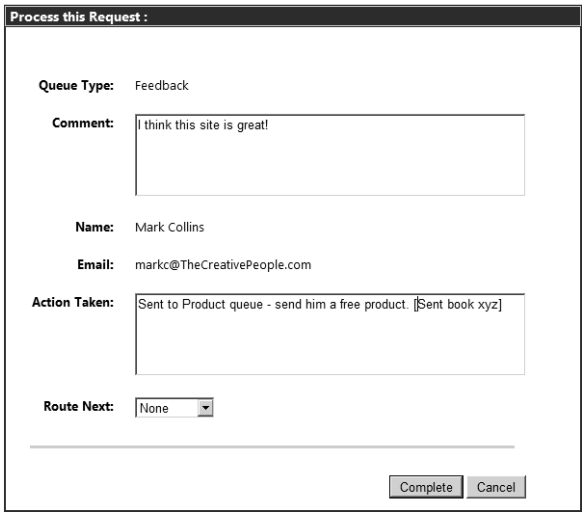


Figure A-10. Working the request in the Product queue

Because there is only one item in the Product queue, it was automatically assigned. You can edit or append to the Action Taken field. You can also reroute this request to another queue, if necessary. For this request, append a note to the Action Taken field and leave the Route Next field blank. Click the Complete button. The page should look like the one shown in Figure A-11.

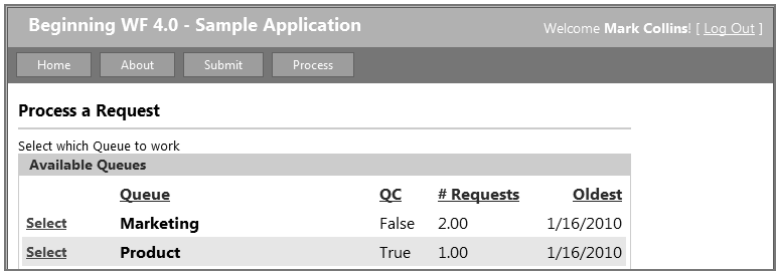


Figure A-11. Request moved to QC mode

Notice that this request is still in the Product queue, but the QC column shows "True". The Product queue is configured so that all requests in the Product queue must go through a QC review. Consequently, the request is put back into the queue in QC mode. Select the Product queue again and the request should be displayed; this time in QC mode, as shown in Figure A-12.

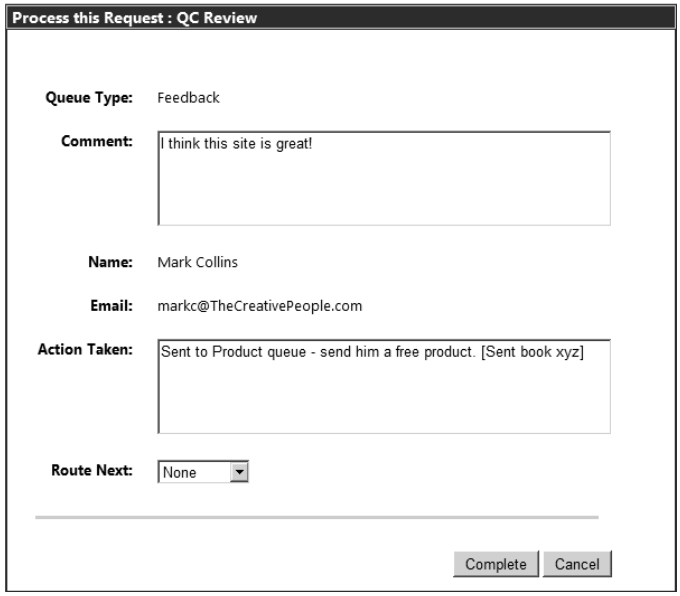


Figure A-12. Performing QC review

In QC review, you can view and modify the Action Taken. Edit this field and click the Complete button. The request is now complete. Figure A-13 shows the updated queue list with only the remaining items in the Marketing queue.

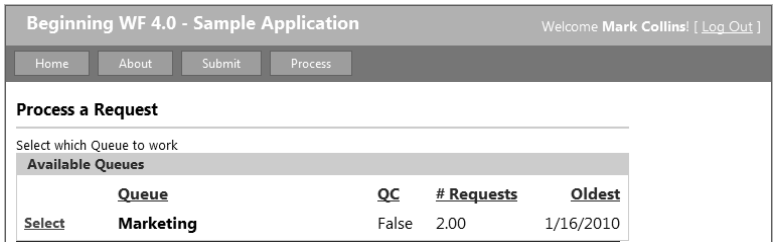


Figure A-13. Updated queue list

Tracking the Workflow

Close the web application and go back to Visual Studio. From the Server Explorer, open the contents of the `QueueTrack` table. The results should be similar to those shown in Figure A-14.

	QueueName	SubQueueN...	QueueInsta...	EventType	QC	OperatorKey	EventDate
▶	Request	Marketing	efcbab4c-4b...	Start	False	*NULL*	1/9/2010 5:...
	Request	Marketing	efcbab4c-4b...	Assign	False	b4a8b43c-e...	1/9/2010 5:...
	Request	Product	efcbab4c-4b...	Route	False	*NULL*	1/9/2010 5:...
	Request	Product	efcbab4c-4b...	Assign	False	b4a8b43c-e...	1/9/2010 5:...
	Request	Product	efcbab4c-4b...	QC	False	*NULL*	1/9/2010 5:...
	Request	Product	efcbab4c-4b...	Assign	True	b4a8b43c-e...	1/9/2010 5:...
	Request	Marketing	3c97677a-c...	Start	False	*NULL*	1/9/2010 5:...
	Request	Marketing	4c41ce42-f8...	Start	False	*NULL*	1/9/2010 5:...

Figure A-14. Showing the tracking results

The `QueueTrack` table records the various events that occurred on each of the requests: Started, Assigned, Route, QC, and so on. The request that you just worked was started in the Marketing queue. It was then assigned to an operator and then routed to the Product queue. It was again assigned to the same operator and then put into QC mode. Finally, it was assigned to the operator for a third time this time in QC mode.

Generic Queue Logic

Using queues for managing human tasks is a common practice that can be used in many applications. I designed this solution to encapsulate the generic activities in a separate project called UserTasks. This should help you to reuse this logic more easily in your own applications.

Database Design

Figure A-15 shows the tables used by the UserTasks project. You can also view this diagram by opening the `UserTasks.dbml` file in the UserTasks project.

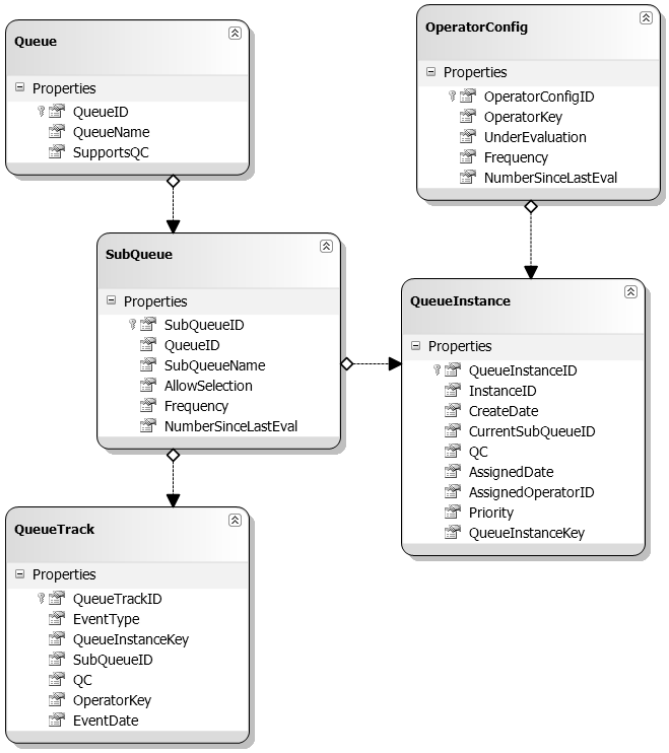


Figure A-15. Database design for the UserTasks project

The database contains both a `Queue` and a `SubQueue` table. The queues that you used, such as Marketing and Product, are really subqueues, and this solution uses a single queue called Request. This approach allows you to reuse the same tables (and workflow activities) for any number of human-centric workflow tasks.

The `Queue` and `SubQueue` tables provide configuration options such as `SupportsQC` (at the queue level) and `AllowSelection` (at the subqueue level). The `SubQueue` table includes the `Frequency` column, which defines how often requests in this subqueue need to be reviewed. The `NumberSinceLastEval` is used to keep track of this to know when it's time to force another review. The `OperatorConfig` table provides other QC-related options (see Chapter 20 for more details.)

The `QueueInstance` table is the main table that drives the queue logic. A record is created for every request. It keeps track of what subqueue the request is currently in, whether it's in QC mode, and who it is currently assigned to. The `QueueTrack` table is populated by the tracking extension in response to user-defined tracking events.

Activities

The UserTasks project is an activity library project that provides activities that can be dropped onto other workflows. The included activities are listed in Table A-1.

Table A-1. Activities provided in the UserTasks project

Activity Name	Description
`AssignQueue`	Moves a `QueueInstance` to the specified subqueue.
`AssignQueueInstance`	Assigns a `QueueInstance` to the specified operator.
`CompleteInstance`	Provides QC and rerouting logic.
`CreateQueueInstance`	Creates a new `QueueInstance` record.
`GetQueueInstances`	Returns the available `QueueInstance` records in the specified subqueue (if allow selection is turned off, it returns only the oldest record).
`LoadQueueInstance`	Loads the specified `QueueInstance`.
`LookupQueueStats`	Returns the number of records in each queue/subqueue.
`RequestQC`	Moves the specified `QueueInstance` into QC mode.
`UnAssignQueueInstance`	Unassigns the current operator and makes this `QueueInstance` available to be assigned to another operator.

All these activities, with the exception of `CompleteInstance`, are implemented as coded activities. I won't take the time to list all the source code here; you can browse the code in Visual Studio. They use a `DBConnection` extension to obtain the connections string (see Chapter 12) and use a `PersistQueueInstance` extension to save the changes as part of the workflow persistence (see Chapter 15). As with all the other projects in this book, they use LINQ to SQL to access the database tables.

CompleteInstance

The `CompleteInstance` activity is provides as a designed activity. The workflow design for it is shown in Figure A-16.

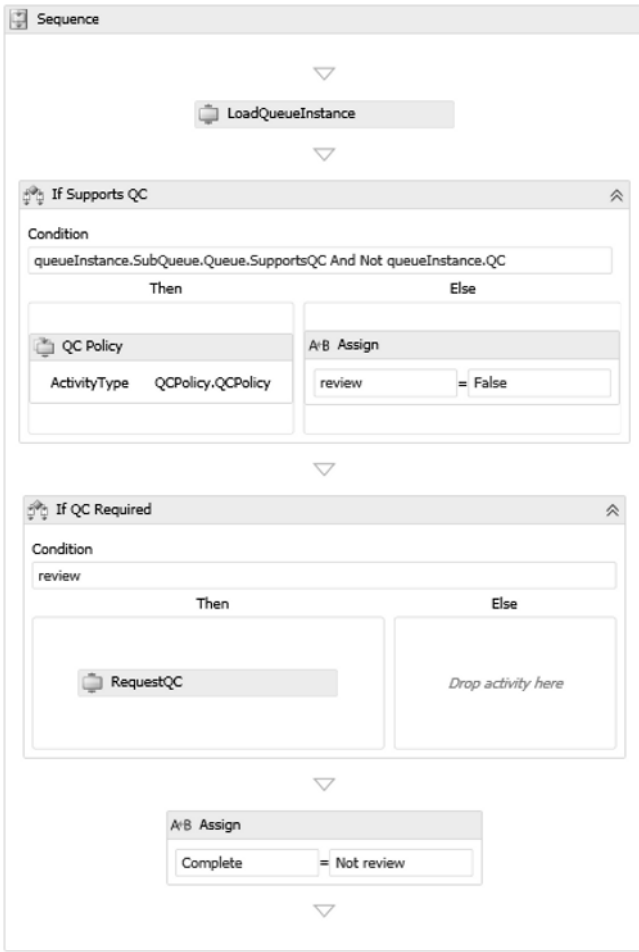


Figure A-16. Design of the CompleteInstance activity

After loading the `QueueInstance` record, it then checks to see whether this queue supports QC or whether the `QueueInstance` is already in QC mode. If not, it invokes the `QCPolicy` activity, which determines whether this record should be QC'ed. If QC is required, the `RequestQC` activity is executed to update the `QueueInstance`. The `Complete` output argument is passed back to the calling workflow to indicate whether the `QueueInstance` was actually completed or whether it requires a QC step.

QCPolicy

The `QCPolicy` activity is an `Interop` activity that invokes a `QCPolicy` activity, which is implemented in .Net 3.5. (You might want to refer to Chapter 20 for more details on using the `Policy` activity.) The design of the `QCPolicy` activity is shown in Figure A-17.

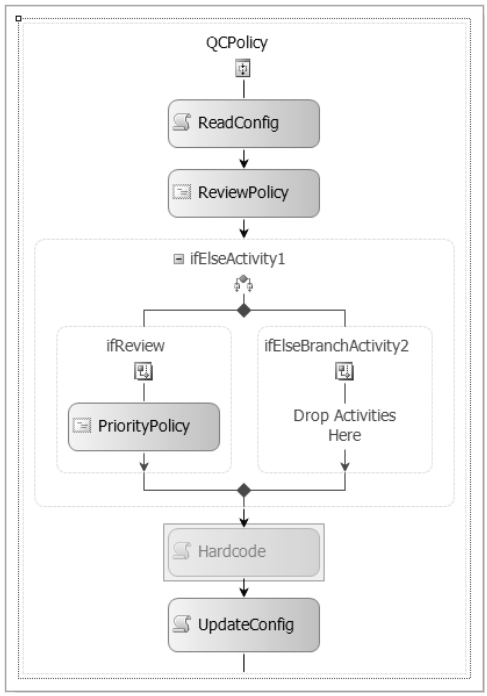


Figure A-17. QCPolicy Design (.Net 3.5)

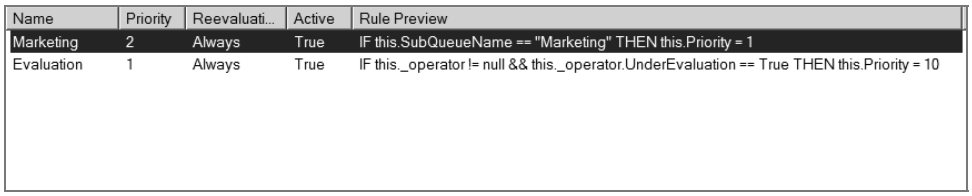
Unlike the implementation in Chapter 20, this `QCPolicy` activity reads and updates the configuration data from the database. Because the `Policy` activity uses the .Net 3.5 version of workflow, it cannot access the `DBConnection` extension. Instead, the connection string is passed in as a `DependencyProperty`.

The `ReviewPolicy` activity is a `Policy` that determines whether the `QueueInstance` needs to be reviewed. The rule set used is shown in Figure A-18.

Name	Priority	Reeval...	Active	Rule Preview
No Configuration	5	Always	True	IF this._queue == null THEN this.Review = False Halt
ActivityFrequency	4	Always	True	IF this._queue.NumberSinceLastEval >= this._queue.Frequency THEN this.Review = True...
No Operator	3	Always	True	IF this._operator == null THEN this.Review = False Halt
Under Evaluation	2	Always	True	IF this._operator.UnderEvaluation == True THEN this.Review = True Halt
Operator Frequency	1	Always	True	IF this._operator.NumberSinceLastEval >= this._operator.Frequency THEN this.Review = ...

Figure A-18. ReviewPolicy rule set

If QC is needed, the `PriorityPolicy` is executed to determine what priority it should be given. Its rule set is shown in Figure A-19.



Name	Priority	Reevaluati...	Active	Rule Preview
Marketing	2	Always	True	IF this.SubQueueName == "Marketing" THEN this.Priority = 1
Evaluation	1	Always	True	IF this._operator != null && this._operator.UnderEvaluation == True THEN this.Priority = 10

Figure A-19. PriorityPolicy rule set

The `Hardcode` activity is used for testing only and is normally disabled. It overrides the `Review` and `Priority` properties that were set by the `Policy` activities. This was provided to make it easier to test both QC and non-QC scenarios.

■ **Note** WF 3.0/3.5 provided the ability to disable an activity. When the workflow is executed, disabled activities are ignored. This feature had limited usefulness and is *not* provided in WF 4.0. This particular scenario was one of the useful applications of this feature.

To test the actual `Policy` implementation, a separate TestQC application is provided. This is a simple workflow application with the `QCPolicy` activity dropped onto it.

Tracking

The ability to track workflow events was described in Chapter 13. This project relies on custom tracking events. The following code (or something similar) is included in several of the custom activities:

```
CustomTrackingRecord userRecord = new CustomTrackingRecord("Route")
{
    Data =
    {
        {"QueueInstanceKey", qi.QueueInstanceKey},
        {"SubQueueID", qi.CurrentSubQueueID}
    }
};

// Emit the custom tracking record
context.Track(userRecord);
```

This causes a custom tracking event to be generated, which is received and processed by the `QueueTracking` extension. The implementation of the `Track` method of this extension is shown in Listing A-1.

Listing A-1. Implementation of the Track Method

```
protected override void Track(TrackingRecord record, TimeSpan timeout)
{
    CustomTrackingRecord customTrackingRecord =
        record as CustomTrackingRecord;

    if (customTrackingRecord != null)
    {
        if (customTrackingRecord.Name == "Start"    ||
            customTrackingRecord.Name == "Route"    ||
            customTrackingRecord.Name == "Assign"   ||
            customTrackingRecord.Name == "UnAssign" ||
            customTrackingRecord.Name == "QC")
        {
            QueueTrack t = new QueueTrack();

            // Extract all the user data
            if ((customTrackingRecord != null) &&
                (customTrackingRecord.Data.Count > 0))
            {
                foreach (string key in customTrackingRecord.Data.Keys)
                {
                    switch (key)
                    {
                        case "QueueInstanceKey":
                            if (customTrackingRecord.Data[key] != null)
                                t.QueueInstanceKey =
                                    (Guid)customTrackingRecord.Data[key];
                            break;
                        case "SubQueueID":
                            if (customTrackingRecord.Data[key] != null)
                                t.SubQueueID = (int)customTrackingRecord.Data[key];
                            break;
                        case "QC":
                            if (customTrackingRecord.Data[key] != null)
                                t.QC = (bool)customTrackingRecord.Data[key];
                            break;
                        case "OperatorKey":
                            if (customTrackingRecord.Data[key] != null)
                                t.OperatorKey =
                                    (Guid)customTrackingRecord.Data[key];
                            break;
                    }
                }
            }

            if (t.SubQueueID != null && t.QC == null)
                t.QC = false;
```

```
            t.EventType = customTrackingRecord.Name;
            t.EventDate = DateTime.UtcNow;

            // Insert a record into the TrackUser table
            UserTasksDataContext dc =
                new UserTasksDataContext(_connectionString);
            dc.QueueTracks.InsertOnSubmit(t);
            dc.SubmitChanges();
        }
    }
}
```

This first checks to see whether this is one of the events that should be processed. Specifically, it is looking for the following:

- `Start`: A new `QueueInstance` is created.
- `Route`: A `QueueInstance` is placed in a queue.
- `Assign`: The `QueueInstance` is assigned to a specific operator.
- `UnAssign`: The `QueueInstance` is unassigned.
- `QC`: The `QueueInstance` is placed to QC mode.

■ **Note** As I demonstrated in Chapter 13, the events that should be tracked can be configured instead of written in code. This is the preferred approach. However, this tracking extension is writing to a database table specifically designed for these events. It doesn't know how to track other events. In this case, it is appropriate to put the filter in code. This still allows you to configure the events that are actually tracked within this set of supported events.

The various data records are extracted from the event to populate the corresponding columns in the table.

Service Layer

As I mentioned previously, all the workflow functionality is provided by a web service. You created a fairly simple web service in Chapter 10. This implementation is significantly more complex.

Service Contract

In Chapter 10, you used both the traditional method of defining a service contract and a declarative style provided by WF. In this project, I used the later method exclusively. On each of the `Receive` activities, the `OperationName` and appropriate input parameters are defined. Similarly, on the `SendReply` activities the output parameters are defined.

Also, on the `Receive` activities, the interface name is specified. You don't have to define this interface; just specify the name to use when one is created for you. If you want all the methods on the same interface, use the same interface name on all the `Receive` activities. You can also use different names, which will result in multiple interfaces. For this project, I used `IProcessRequest` for all the methods. Figure A-20 shows the methods that are implemented in the `IProcessRequest` interface.

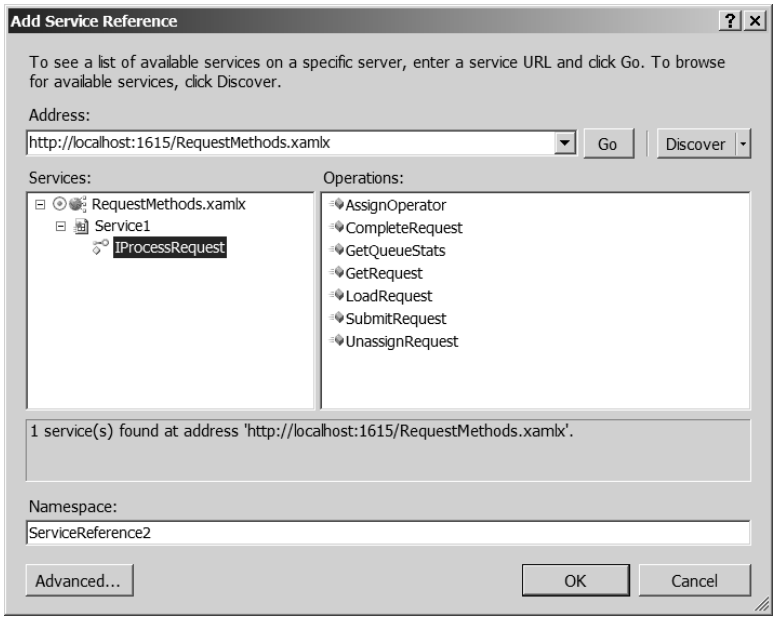


Figure A-20. Web service methods

When a service reference is added to the web application, all the necessary web service details are generated for you, including the Web Service Definition Language (`.wsdl`) file that explicitly defines the web service and the methods provided. You can open `Service1.wsdl` and see what this looks like. It's a little cryptic for the human reader. It also generates several `.xsd` files that define the data types and data contracts that are used in the input and output messages. Figure A-21 shows one of these files displayed in the XML Schema Explorer.

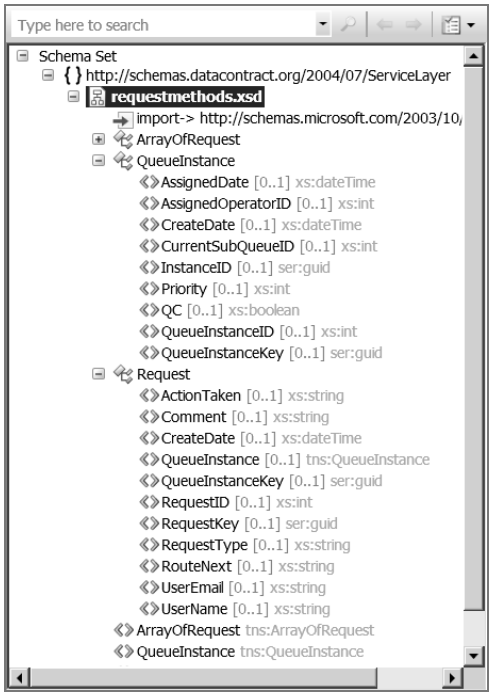


Figure A-21. XML Schema Explorer

Database Design

Because the queue logic is provided by the UserTasks project, the service layer can focus on request-specific design elements. Consequently, the data model is quite simple, as shown in Figure A-22.

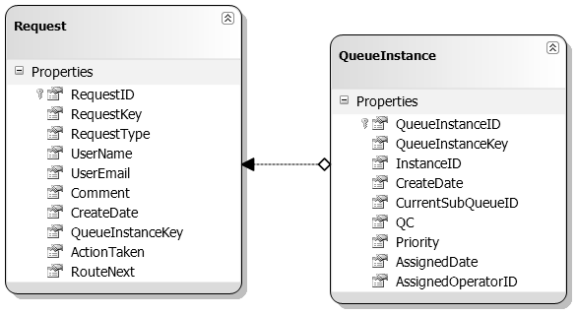


Figure A-22. Service layer data model

The Request table contains the data, such as name, e-mail, and comment, entered by the user who submitted the request. It also records the action taken, which is entered by the operator who worked the request. It has a reference to a QueueInstance record. The QueueInstance record handles all the queue details such as the current queue, to whom it's assigned, and so on. This design keeps the Request table clear of all these "plumbing" details.

Activities

The ServiceLayer project implements some custom activities, which are listed in Table A-2.

Table A-2. Activities provided in the ServiceLayer project

Activity Name	Description
BuildRequestList	Maps the data returned by GetQueueInstances into a list of Request objects.
CreateRequest	Creates a new Request record.
LoadRequest	Load the specified Request from the database.
UpdateRequest	Updates a Request.

The CreateRequest and UpdateRequest use the PersistRequest extension to perform the database update when the workflow is persisted. The BuildRequestList activity is the only one that is particularly interesting. The GetQueueInstances activity (provided in the UserTasks project) handles all the logic to determine which records are available to be worked, but it knows nothing about requests. It uses only the QueueInstance table as well as the Queue and SubQueue setup tables. BuildRequestList takes the list of QueueInstance objects returned by GetQueueInstances and maps them to a list of Request objects. The implementation is shown in Listing A-2.

Listing A-2. BuildRequestList Implementation

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Activities;
using UserTasks;
using UserTasks.Extensions;

namespace ServiceLayer.Activities
{

    public sealed class BuildRequestList : CodeActivity
    {
```

```
        public InArgument<UserTasks.QueueInstance[]> QueueInstanceList { get; set;}
        public OutArgument<Request[]> RequestList { get; set; }

        protected override void Execute(CodeActivityContext context)
        {
            // Get the connection string
            DBConnection ext = context.GetExtension<DBConnection>();
            if (ext == null)
               throw new InvalidProgramException("No connection string available");

            RequestDataContext dc = new RequestDataContext(ext.ConnectionString);

            // Get the list of QueueInstances
            UserTasks.QueueInstance[] qiList = QueueInstanceList.Get(context);
            if (qiList != null && qiList.Count() > 0)
            {
                // Build a list of Request objects
                Request[] rList = new Request[qiList.Count()];
                int i = 0;
                foreach (UserTasks.QueueInstance qi in
                    QueueInstanceList.Get(context))
                {
                    Request r = dc.Requests.SingleOrDefault
                        (x => x.QueueInstanceKey == qi.QueueInstanceKey);
                    rList[i++] = r;
                }

                RequestList.Set(context, rList);
            }
        }
    }
}
```

Workflow Design

This web service is implemented using the workflow designer, which produces an `.xamlx` file. Figure A-23 shows the overall design with some of the activities collapsed to fit the diagram on a page.

Figure A-23. Overall workflow design

The workflow uses the `Pick` activity that was introduced in Chapter 10. To review, a `Pick` activity contains one or more `PickBranch` activities, in which each contains a Trigger activity and an Action sequence. When the Trigger is executed, the associated Action sequence is started, and all other `PickBranch` activities are cancelled. In this case there are four branches; the Trigger for each one contains a `Receive` activity that waits for a specific web service method.

The four methods are as follows:

- `SubmitRequest`: Initiates a new request.
- `GetQueueStats`: Gets the number of request in each queue.
- `GetRequest`: Gets the available request(s) from the specified queue.
- `LoadRequest`: Returns the request details of the specified request.

The last three are fairly straightforward. They perform a database operation and return the appropriate results back to the caller using a `SendReply` activity. When the application wants to get the current statistics about the queues, it calls the `GetQueueStats()` method of the web service. This will create a new workflow instance that is completed as soon as the response is sent back to the application. This workflow is very short-lived. The instance store is configured to remove the record from the `InstancesTable` when the workflow is completed. So using a workflow instance to perform a simple task does not leave any artifacts behind.

SubmitRequest

When `SubmitRequest` is called, it first determines the correct subqueue using a `Switch` activity (see Chapter 4 for details). It then creates a `QueueInstance` record and a `Request` record and returns data back

to the caller using `SendReply`. So far, this is similar to the other branches. However, the workflow continues after the `SendReply` activity. It is followed by a `While` activity; the design of this is shown in Figure A-24.

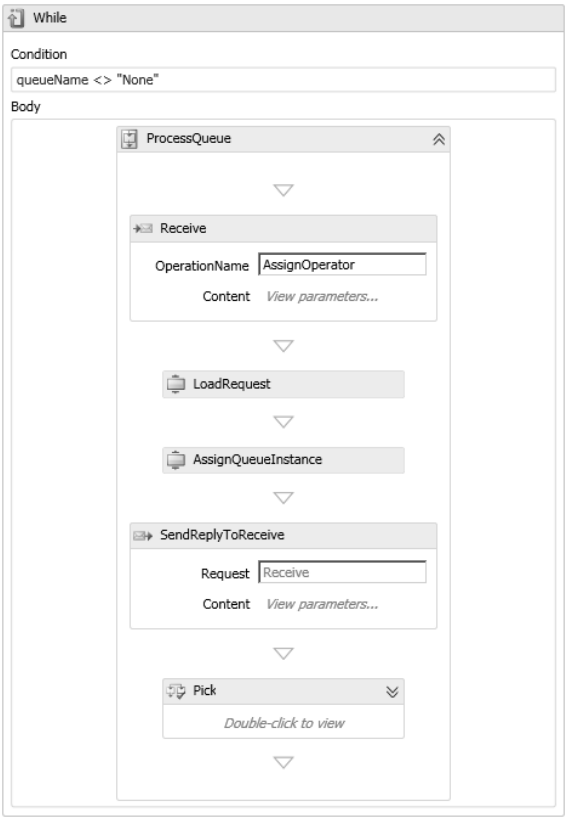


Figure A-24. While activity

The `Condition` on the `While` activity is as follows:

```
queueName <> "None"
```

This means that the `While` activity will continue to execute as long as the request is assigned to a queue. When the request is completed, without forwarding the request to another queue, the `queueName` is set to "None", causing the `While` activity to complete. The logic inside the `While` activity starts with a `Receive` activity that waits for the `AssignOperator` method. A request must be assigned to an operator before it can be worked. This helps ensure that two people are not working on the same request.

Processing a Request

When the `AssignOperator()` method is called, the `QueueInstance` associated with that request is updated to record the assignment and the response is sent back. The workflow then continues with another `Pick` activity that is shown in Figure A-25.

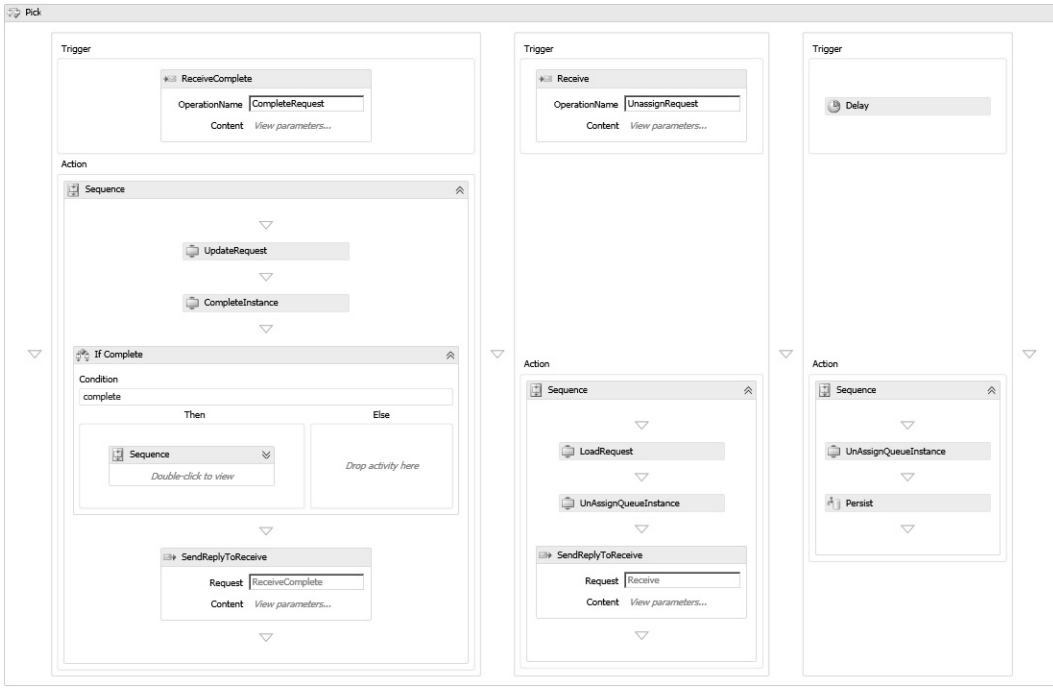


Figure A-25. The final Pick activity

When a request has been assigned to an operator, one of three things can happen:

- The request is completed.
- The page is cancelled forcing the request to be unassigned.
- Neither.

The three branches represent these scenarios. The first branch completes the request, and the second branch unassigns the request so someone else can work it. The third branch uses a `Delay` activity to wait for five minutes. If nothing has been done within that time, the request is automatically unassigned. If an operator has a request assigned to them and they decide to leave for the day and simply close their browser, the request would be left assigned to them. This means that no one else could work that request. This third branch was added to take care of that scenario.

If the `CompleteRequest()` method is called, the first branch updates the request with the data provided. It then executes the `CompleteInstance` activity that was described earlier. This activity

determines whether the item in the queue needs to be reviewed in QC mode. It outputs a `Complete` argument to indicate whether the item is complete or whether it needs to be reviewed. Figure A-26 shows the sequence that is executed if the request is complete.

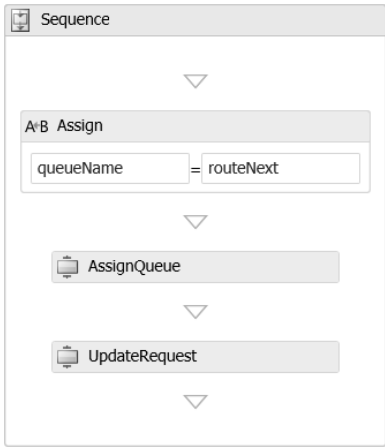


Figure A-26. The sequence for completed requests

■ **Note** When I say the request is complete, I'm referring to that particular task for that request is complete. The request might need to be worked in other queues before the workflow is completed.

The `routeNext` variable is specified in the `CompleteRequest()` method; it is determined by the operator working the request. This is copied to the `queueName` variable. If no queue was selected, this will signal that the workflow is done, and the `While` activity will complete. The `AssignQueue` activity is called to update the current subqueue for the `QueueInstance`. The `UpdateRequest` activity will clear the `RouteNext` field.

Correlation

The concept of correlation was introduced briefly in Chapter 8. A typical workflow will have hundreds, perhaps thousands, of workflow instances executing simultaneously. When a workflow design includes sending messages to (and between) instances, correlation provides the mechanism to ensure that messages are sent to the correct instance. There are three types of correlation provided by WF 4.0.

The first (and probably easiest) correlation is called request reply correlation. It is used when you have a two-way communication between workflow activities. For example, you send a request and wait for the response. You used this in Chapter 8 (and others). By placing the `Send` and `ReceiveReply` activities within a `CorrelationScope` activity, the workflow took care of the details for you. In this case, correlation was accomplished through the communication channel that was established between the sender and the receiver.

The other two types support more complex scenarios in which there are multiple messages between the workflows. The first is referred to as context correlation. In this case, the client sends a request, and the server includes a context ID with the response. This context ID must be included with all subsequent messages from the client. This is used by the server to associate the same instance that responded to the first message. This approach requires logic on both the client and server. This approach also requires that the first message be a two-way message; the server has to send back a response that includes the context ID.

The last approach, query correlation, is accomplished on the server side only. In a sense, this is actually very similar to context correlation. There is some sort of key that is included with each request that identifies the corresponding workflow instance. With context correlation, this key is generated by the server on the initial request. However, with query correlation, this key is based on data included in the message.

This project uses query correlation and the common key is the `RequestKey`, which is a `Guid` generated by the application. The `RequestKey` is provided as one of the parameters on every call to the service. When the first message is received, the `RequestKey` is mapped to that workflow instance. This is done through a correlation initializer, which sets up the mapping between the `RequestKey` and the associated workflow instance. On subsequent calls, this is extracted (queried) from the data in the incoming message and the mapping is used to determine the workflow instance.

Query correlation is accomplished through the `Receive` activity. There are three properties on the `Receive` activity that support correlation, as shown in Figure A-27.

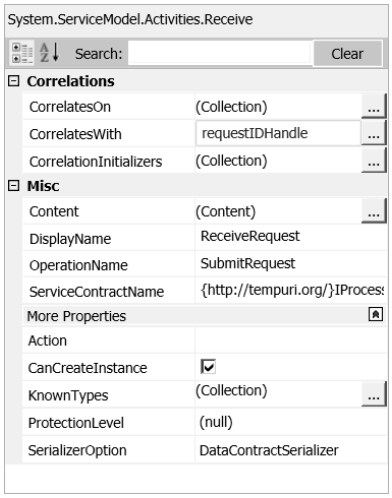


Figure A-27. Properties of a Receive activity

The `CorrelatesWith` property defines the handle, which is specified as a `CorrelationHandle` type. This is a workflow variable that is persisted with the workflow. A `CorrelationInitializer` is then added to the first `Receive` activity, as shown in Figure A-28.

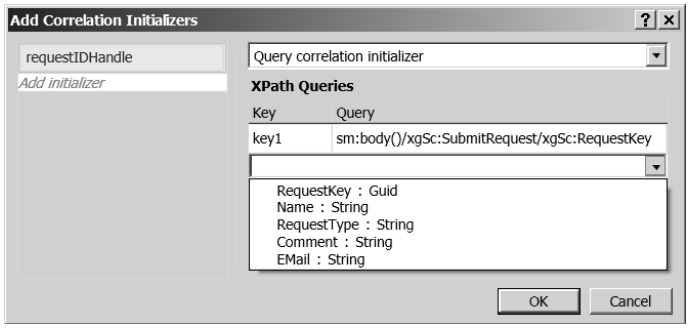


Figure A-28. A correlation initializer

The XPath query might seem a little cryptic, but don't worry; the Visual Studio takes care of this for you. You only need to select the appropriate property from the drop-down menu. The drop-down menu lists all the parameters in the incoming message. When the `RequestKey` is chosen, the query is generated automatically.

On subsequent `Receive` activities, instead of setting the `CorrelationInitializer`, the `CorrelatedOn` property is set. This is done by selecting the correct parameter; a query is then generated for you, as shown in Figure A-29.

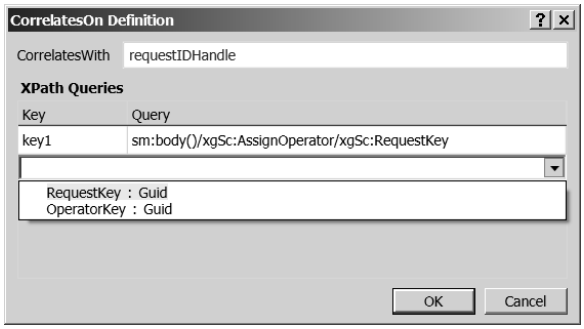


Figure A-29. The CorrelatesOn property

Using WorkflowServiceHost

This project uses several workflow extensions that were introduced in previous chapters (persistence in Chapter 11, sharing configuration data in Chapter 12, tracking in Chapter13, and custom persistence in Chapter 14). In those projects there was a console or WPF application that configured these extensions and added them to the workflow instances as they were created. In this project, this is done by the `WorkflowServiceHost`.

Writing Extensions

To add extensions when using the `WorkflowServiceHost`, they must be configured in the `web.config` or `app.config` file. This requires some extra steps when writing the extensions. The modified implementation of `DBConnection.cs` is shown in Listing A-3.

Listing A-3. Implementation of the DBConnection Extension

```
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.Collections.Specialized;
using System.Configuration;
using System.Web.Configuration;
using System.ServiceModel;
using System.ServiceModel.Activities;
using System.ServiceModel.Channels;
using System.ServiceModel.Configuration;
using System.ServiceModel.Description;

namespace UserTasks.Extensions
{
    /*****************************************************/
    // The extension class is used to define the behavior
    /*****************************************************/
    public class DBConnectionExtension : BehaviorExtensionElement
    {
        public DBConnectionExtension()
        {
            Console.WriteLine("Behavior extension started");
        }

        [ConfigurationProperty("connectionStringName", DefaultValue = "",
            IsKey = false, IsRequired = true)]
        public string ConnectionStringName
        {
            get { return (string)this["connectionStringName"]; }
            set { this["connectionStringName"] = value; }
        }

        public string ConnectionString
        {
            get
            {
                ConnectionStringSettingsCollection connectionStrings =
                    WebConfigurationManager.ConnectionStrings;
                if (connectionStrings == null) return null;
                string connectionString = null;
                if (connectionStrings[ConnectionStringName] != null)
                {
```

```
                connectionString =
                    connectionStrings[ConnectionStringName].ConnectionString;
            }
            if (connectionString == null)
            {
                throw new ConfigurationErrorsException
                    ("Connection string is required");
            }
            return connectionString;
        }
    }

    public override Type BehaviorType
    {
        get { return typeof(DBConnectionBehavior); }
    }
    protected override object CreateBehavior()
    {
        return new DBConnectionBehavior(ConnectionString);
    }
}

/*****************************************************/
// The behavior class is used to create an extension
// for each new instance
/*****************************************************/
public class DBConnectionBehavior : IServiceBehavior
{
    string _connectionString;

    public DBConnectionBehavior(string connectionString)
    {
        this._connectionString = connectionString;
    }

    public virtual void ApplyDispatchBehavior
       (ServiceDescription serviceDescription, ServiceHostBase serviceHostBase)
    {
        WorkflowServiceHost workflowServiceHost
            = serviceHostBase as WorkflowServiceHost;
        if (null != workflowServiceHost)
        {
            string workflowDisplayName
                = workflowServiceHost.Activity.DisplayName;

            workflowServiceHost.WorkflowExtensions.Add(()
                    => new DBConnection(_connectionString));
        }
    }

    public virtual void AddBindingParameters
```

```
            (ServiceDescription serviceDescription,
             ServiceHostBase serviceHostBase,
             Collection<ServiceEndpoint> endpoints,
             BindingParameterCollection bindingParameters)
        {
        }

        public virtual void Validate
            (ServiceDescription serviceDescription,
             ServiceHostBase serviceHostBase)
        {
        }
    }

    /*****************************************************/
    // This is the actual extension class
    /*****************************************************/
    public class DBConnection
    {
        private string _connectionString = "";

        public DBConnection(string connectionString)
        {
            _connectionString = connectionString;
        }

        public string ConnectionString { get { return _connectionString; } }
    }
}
```

There are three classes implemented in this file:

- `DBConnectionExtension`
- `DBConnectionBehavior`
- `DBConnection`

`DBConnectionExtension` is derived from the `BehaviorExtensionElement` class. It specifies the configuration values that are supported. In this case, there is only one: `connectionStringName`. It also provides a `ConnectionString()` method that obtains the connection string from the configuration file using the `connsctionStringName` parameter. Finally, it overrides the `CreateBehavior()` method, which creates a `DBConnectionBehavior` object passing in the connection string to the constructor.

The `DBConnectionBehavior` class implements the `IServiceBehavior` interface. This interface defines an `ApplyDispatchBehavior()` method that creates an extension and adds it to a workflow instance. This is roughly equivalent to the `SetupInstance()` method you wrote in Chapter 12. When the `WorkflowServiceHost` is started, it looks for all the configured extensions and obtains an `IServiceBehavior` interface for each. As each workflow instance is created, it calls the `ApplyDispatchBehavior()` method on each of the `IServiceBehavior` interfaces. The `ApplyDispatchBehavior()` method creates a `DBConnection` class, passing in the connection string to the constructor and then adds it to the `WorkflowExtensions` collection.

The DBConnection class in same implementation you created in Chapter 12. It provides a ConnectionString property that supplies the connection string to any activity that needs it.

Configuring Extensions

A subset of the web.config file is shown in Listing A-4.

Listing A-4. A Portion of the web.config file

```
<configuration>
  <connectionStrings>
    <add name="Request" connectionString=
      "Data Source=localhost;Initial Catalog=Appendix;Integrated Security=True"
      providerName="System.Data.SqlClient" />
  </connectionStrings>
  <system.serviceModel>
    <extensions>
      <behaviorExtensions>
        <add name="dbConnection"
          type="UserTasks.Extensions.DBConnectionExtension, UserTasks,
                Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" />
      </behaviorExtensions>
    </extensions>
    <behaviors>
      <serviceBehaviors>
        <behavior>
          <dbConnection connectionStringName="Request"/>
        </behavior>
      </serviceBehaviors>
    </behaviors>
  </system.serviceModel>
</configuration>
```

First, a connection string named “Request” is defined in the connectionStrings section. This allows you to reference it by name in various places in the web.config file. The advantage to this approach is that the actual connection string is specified only once. If you need to modify it later, you have to change it in only one place.

Next, an extension named “dbExtension” is added to the behaviorExtensions section. Note that the actual class that is referenced is the DBConnectionExtension class, not the DBConnectionBehavior or DBConnection classes. This extension is then configured in the behaviors section. The extension is specified by name, dbConnection, and its configuration values are defined. There is only one, connectionStringName and it is set to “Request” to use the connection string defined earlier.

Configuring Persistence

The persistence extension, SqlWorkflowInstanceStore, is configured in the behavior section as well. You do not need to add anything to the behaviorExtensions section. The subset of the web.config file is shown in Listing A-5.

Listing A-5. Configuring Persistence

```
<behaviors>
  <serviceBehaviors>
    <behavior>
      <sqlWorkflowInstanceStore
        connectionStringName="Request"
        instanceCompletionAction="DeleteAll"
        instanceLockedExceptionAction="NoRetry"
        instanceEncodingOption="GZip"
        hostLockRenewalPeriod="00:00:30" />
      <workflowIdle
        timeToUnload="00:00:10"
        timeToPersist="00:00:05" />
    </behavior>
  </serviceBehaviors>
</behaviors>
```

Notice that is uses the same `connectionStringName`. The code in Listing A-5 also configures the `workflowIdle` behavior. The `timeToUnload` property is set to 10 seconds. This will keep the instance in memory for 10 seconds after it has entered the `Idle` state.

Configuring Tracking

To add the tracking extension `QueueTracking`, the entries are added to extensions and behavior sections just as it was for the `DBExtension` discussed previously. In addition, a tracking section is added to specify queries used to define the tracking events that are to be included. Refer to Chapter 13 for more information about tracking queries. The `web.config` entries are shown in Listing A-6.

Listing A-6. Configuring Tracking

```
<configuration>
  <system.serviceModel>
    <extensions>
      <behaviorExtensions>
        <add name="tracking"
          type="UserTasks.Extensions.QueueTrackingExtension, UserTasks,
            Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" />
      </behaviorExtensions>
    </extensions>
    <behaviors>
      <serviceBehaviors>
        <behavior>
          <tracking connectionStringName="Request"/>
        </behavior>
      </serviceBehaviors>
    </behaviors>
    <tracking>
      <profiles>
```

```
        <trackingProfile name="Queue_Tracking">
          <workflow>
            <customTrackingQueries>
              <customTrackingQuery name="*" activityName="*" />
            </customTrackingQueries>
          </workflow>
        </trackingProfile>
      </profiles>
    </tracking>
  </system.serviceModel>
</configuration>
```

The complete web.config file is shown in Listing A-7.

Listing A-7. Complete web.config File

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <connectionStrings>
    <add name="Request" connectionString=
      "Data Source=localhost;Initial Catalog=Appendix;Integrated Security=True"
      providerName="System.Data.SqlClient" />
  </connectionStrings>
  <system.web>
    <compilation debug="true" targetFramework="4.0" />
  </system.web>
  <system.serviceModel>
    <extensions>
      <behaviorExtensions>
        <add name="persistRequest"
          type="ServiceLayer.Extensions.PersistRequestExtension, ServiceLayer,
            Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" />
        <add name="persistQueueInstance"
          type="UserTasks.Extensions.PersistQueueInstanceExtension, UserTasks,
            Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" />
        <add name="dbConnection"
          type="UserTasks.Extensions.DBConnectionExtension, UserTasks,
            Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" />
        <add name="tracking"
          type="UserTasks.Extensions.QueueTrackingExtension, UserTasks,
            Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" />
      </behaviorExtensions>
    </extensions>
    <behaviors>
      <serviceBehaviors>
        <behavior>
          <!-- To avoid disclosing metadata information, set the value below to
               false and remove the metadata endpoint above before deployment -->
          <serviceMetadata httpGetEnabled="True"/>
          <!-- To receive exception details in faults for debugging purposes,
               set the value below to true.  Set to false before deployment to
```

```
            avoid disclosing exception information -->
        <serviceDebug includeExceptionDetailInFaults="True"/>
        <!-- This line configures the persistence service -->
        <sqlWorkflowInstanceStore
          connectionStringName="Request"
          instanceCompletionAction="DeleteAll"
          instanceLockedExceptionAction="NoRetry"
          instanceEncodingOption="GZip"
          hostLockRenewalPeriod="00:00:30" />
        <workflowIdle
          timeToUnload="00:30:00"
          timeToPersist="00:00:05" />
        <!-- Configure the connection string for the persistence extensions-->
        <dbConnection connectionStringName="Request"/>
        <persistRequest connectionStringName="Request"/>
        <persistQueueInstance connectionStringName="Request"/>
        <tracking connectionStringName="Request"/>
      </behavior>
    </serviceBehaviors>
  </behaviors>
  <tracking>
    <profiles>
      <trackingProfile name="Queue_Tracking">
        <workflow>
          <customTrackingQueries>
            <customTrackingQuery name="*" activityName="*" />
          </customTrackingQueries>
        </workflow>
      </trackingProfile>
    </profiles>
  </tracking>
 </system.serviceModel>
 <system.webServer>
  <modules runAllManagedModulesForAllRequests="true"/>
 </system.webServer>
</configuration>
```

Summary

This sample project is just one way workflow can be used to implement a solution. There are other approaches, such as the one described in Chapter 7 in which workflow was used to organize a processing algorithm. My goal throughout this book was to give you a variety of applications. Hopefully one or more of these will resemble a project you are currently working on.

You now have the tools to use the capabilities provided by Workflow Foundation. I wish you great success as you add this to your repertoire of software design patterns.

Index

B

C

D

E

F

G

H

I, J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X, Y

Z