CHAPTER 3

The Simplest Notification Application: Stock Quotes

IN THIS CHAPTER

- The SQL-NS Application Model
- Building the Stock Application's ADF
- Specifying Other Parts of the Stock Application
- Running the Stock Application
- Inside the Running Application
- What Has the SQL-NS Platform Provided?
- Cleaning Up the Instance and Application

In this chapter, you'll see your first SQL-NS application: a stock notification service similar to those offered by many real-world stockbrokers. The application allows subscribers to enter subscriptions for stocks in which they are interested and notifies them when those stocks cross the price targets they specify.

Think of this chapter as a tour. My intent is simply to show you around the various facilities that the SQL-NS platform offers so that you get a feel for the application model and the process of coding to it.

We will look at code in this chapter, but simply for the purpose of understanding the concepts behind the application. A line-by-line explanation of the code at this stage would drown out the simpler picture that I'm trying to show.

Instead, I will gloss over (or in some cases, completely ignore) some parts of the code you will see; I'll just highlight those pieces of the code that illustrate particularly important parts of the application model. Subsequent chapters will cover the rest in detail.

The SQL-NS Application Model

SQL-NS can be used to build a variety of notification applications with different uses and for different application domains. But, when viewed from the

highest level, all these notification applications conform to the same basic model, shown in Figure 3.1.

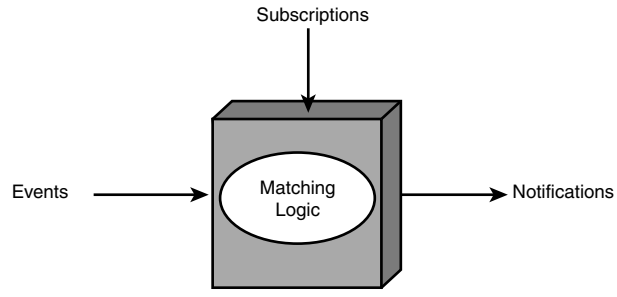

FIGURE 3.1 High-level view of a notification application.

Data enters the application from the outside world. This data can be pulled in by the application or pushed in by an external source. In SQL-NS terms, each piece of data is referred to as an *event* because it represents some happening in the outside world that may potentially be of interest to some subscribers. An event may be a new price for a stock, a notice of a new concert listing, or a gate change for a flight.

The notification application maintains users' subscriptions. Subscriptions are users' declarations of what kinds of events interest them. When events arrive, the application matches them against the subscriptions and produces a set of notifications. These notifications are delivered to the end users.

Events as Data

Events are descriptions of things that happen in the real-world that can be represented as data. For example, a change in the price of a stock (an event potentially of interest to a stockbroker client) can be described as a piece of structured data containing a stock symbol field and a stock price field. A traffic incident event (for an application that notifies commuters about road conditions) might contain a field that describes the location of the incident and another describing the incident type (accident, road closure, weather warning, and so on).

Whatever the type of event, its description can be modeled as data. The structure of the data can be described with a schema that indicates the names of the fields and their data types. Given this schema, it's easy to construct a database table to store the event data. For example, stock events can be stored as rows in a table, as shown in Figure 3.2.

Stock Symbol	Stock Price
XYZ	55.55
PQS	95.30
JKL	15.00

FIGURE 3.2 Modeling stock events as rows in a table.

Subscriptions as Data

Thinking of events as data is usually quite natural. Although it may be somewhat less intuitive at first, subscriptions can be modeled as data, too. Think of the subscriptions in a stock notification application. Let's say that all subscriptions will have the form

"Notify me when the price of stock *S* reaches price target *T*."

where *S* represents some stock symbol and *T* represents a price target. A subscription example might be

"Notify me when the price of stock XYZ reaches price target $50.00."

If all the subscriptions are constrained to this form, an individual subscription can be represented as a pair of values for *S* and *T*. Such subscriptions could be stored in a table, as shown in Figure 3.3.

Subscriber	Stock Symbol (S)	Price Target(T)
Bob	XYZ	50.00
Alex	PQS	90.00
Mary	JKL	12.00
Jane	PQS	100.00

FIGURE 3.3 Modeling stock subscriptions as rows in a table.

Each row identifies the subscriber, the stock symbol, and the price target of interest. For illustrative purposes, the subscriber is represented by a name, but in an actual application, a more appropriate identifier may be used.

Matching Events with Subscriptions

As mentioned in Chapter 1, "An Overview of Notification Applications," the matching of events against subscriptions is the key function of any notification application. If the matching can be implemented efficiently, the application will scale to large volumes.

With events and subscriptions both represented as data, matching can be accomplished by means of a SQL join. Given the table structures in Figures 3.2 and 3.3 for events and subscriptions—and let's say that we called the events table `Events` and the subscriptions table `Subscriptions`—the following SQL statement would determine the matches:

```
SELECT  S.Subscriber, E.StockSymbol, E.StockPrice
FROM    Events E JOIN Subscriptions S
ON      E.StockSymbol = S.StockSymbol
WHERE   E.StockPrice >= S.PriceTarget
```

This statement joins the stock events table with the subscriptions table on stock symbol and then selects rows where the stock price in the event is greater than or equal to the stock price target specified in the subscription. The rows returned by this query represent

the set of matches between the events and subscriptions that should result in notifications being sent. For the particular data shown in the preceding examples, Figure 3.4 shows the results of the query.

Subscriber	Stock Symbol	Stock Price
Bob	XYZ	55.55
Alex	PQS	95.30
Mary	JKL	15.00

FIGURE 3.4 Results of matching events with subscriptions.

Note that only three of the four subscriptions matched: Jane's subscription specified a price target for PQS of 100.00, and because the event for PQS indicated that the price was only 95.30, the matching query did not return a row for Jane.

Each of the rows in the results table is the raw data for a notification to be sent. Thus, notifications also can be modeled as rows of data in a table. The notification data can later be packaged into a readable message and delivered to the appropriate subscriber.

Scalability of the SQL-NS Application Model

The modeling of both events and subscriptions as data is a key innovation of SQL-NS and the basis for its inherent scalability. Because both events and subscriptions are rows in tables, SQL joins can be used to match them. In general, SQL joins are extremely efficient at matching large sets of data; more than 20 years of query processing and indexing developments make this possible. As long as a reasonable join query can be written for a particular event and subscription schema (in most cases, one can), the cost of matching (in terms of computing resources) will be low. Furthermore, this cost grows sublinearly with the amount of data. That is, if you double the number of events or subscriptions, the cost of matching increases by less than a factor of two.

This model is different from that used by most other pub-sub systems. Most other systems model individual subscriptions as queries, rather than data. The simplest of these systems evaluates the subscription queries one-by-one for a given set of events. This strategy is expensive, and as the number of subscriptions and events grows, the cost of evaluation grows—at best, linearly, at worst, exponentially. The more sophisticated of these systems attempt to obtain performance gains by indexing the subscription queries and looking for logical shortcuts in the query evaluation. But the effectiveness of these strategies is limited by the structural differences and complexity of the subscriptions and isn't always assured. In many cases, systems that model subscriptions as queries can do little better than one-at-a-time evaluation.

NOTE

To be absolutely clear, the SQL-NS application model does use queries to evaluate subscriptions. But it does not model *each individual subscription* as a query. Instead, in

the SQL-NS application model, there is one query for each subscription type. This query evaluates *all* subscriptions of that type at once. This is the key differentiator between SQL-NS and other pub-sub systems.

When you build a SQL-NS application, you have a choice of two models for subscription evaluation. In the first model, *developer-defined logic* is used to determine whether a match between an event and a subscription has occurred. In this model, you, the application developer, write the SQL join query that finds the matches. This query might look something like the SQL statement shown earlier in the "Matching Events with Subscriptions" section (p. 43).

By writing the query, you are defining the conditions that must be true for a match to occur. The subscribers can specify only values for the data referenced in the query. For example, the logic coded into the stock application's query says that a stock event matches a stock subscription when the stock symbols are the same and the stock price in the event is greater than, or equal to, the stock price target in the subscription. Subscribers cannot change this logic; they can only provide values for the stock symbol and stock price target in individual subscriptions.

In the second subscription evaluation model, *user-defined logic* determines whether an event matches a subscription. The users of the application (the subscribers) choose the conditions that must be true for a match to occur. These conditions can be arbitrary combinations of Boolean predicates based on the event data. In this model, each subscription can specify a different matching condition. For example, given the stock event data, one subscription's condition might state that a match occurs when the stock price in the event is *less than* a particular price target. Another subscription's condition may stipulate that a match occurs when the stock price in the event is within a particular range of values.

Regardless of whether you choose developer-defined or user-defined logic for the subscriptions in your application, SQL-NS uses queries to evaluate groups of subscriptions at the same time. In the case of developer-defined logic, the query provided by the developer is associated with a named subscription type, and all subscriptions of that type are evaluated by the single query. When user-defined logic is used, SQL-NS also uses a single query to evaluate subscriptions of a common type. The difference is that subscriptions are grouped into types based on the logical structure of their conditions (rather than assigned to predefined, named subscription types), and the queries used to evaluate them are constructed dynamically by SQL-NS. For a description of how SQL-NS uses queries to evaluate subscriptions with user-defined logic, refer to the "Evaluating User-Defined Logic with Queries" sidebar (p. 45).

EVALUATING USER-DEFINED LOGIC WITH QUERIES

To evaluate user-defined logic, SQL-NS begins by translating each subscription into a SQL query that selects from the events table the rows that satisfy the subscription condition. The condition logic specified in the subscription is implemented in the `WHERE` clauses of this query. For example, suppose a subscriber, Bob, wants to be notified when

the XYZ stock price falls between 45 and 75. He might enter a subscription with the following condition:

```
(E.StockSymbol = 'XYZ') AND ((E.StockPrice > 45) AND (E.StockPrice <= 75))
```

This could be translated into the following SQL query:

```
SELECT  'Bob' AS Subscriber, E.StockSymbol, E.StockPrice
FROM    Events E
WHERE   E.StockSymbol = 'XYZ'
        AND (E.StockPrice > 45 AND E.StockPrice <= 75)
```

When two or more subscription conditions translate into SQL queries with the same structure—that is, queries with the same combinations of logical constructs that differ only in terms of constant data values, SQL-NS builds a single query to evaluate them. That query does not contain hard-coded data values (as the one in the preceding example does), but it references a parameters table that stores the specific constant values for each subscription. For example, suppose that in addition to Bob's subscription we just examined, another subscriber, Mary, wants to be notified when the PQS stock price falls between 80 and 120. Mary's subscription condition

```
(E.StockSymbol = 'PQS') AND ((E.StockPrice > 80) AND (E.StockPrice <= 120))
```

could be translated into the following query:

```
SELECT  'Mary' AS Subscriber, E.StockSymbol, E.StockPrice
FROM    Events E
WHERE   E.StockSymbol = 'PQS'
        AND (E.StockPrice > 80 AND E.StockPrice <= 120)
```

But because this query is structurally the same as the query that evaluates Bob's subscription, both subscriptions could be evaluated together by the following single query:

```
SELECT  P.Subscriber, E.StockSymbol, E.StockPrice
FROM    Events E, SubscriptionParameters P
WHERE   E.StockSymbol = P.Parameter1
        AND (E.StockPrice > P.Parameter2 AND E.StockPrice <= P.Parameter3)
```

This query relies on the `SubscriptionParameters` table to provide the constant data values for each subscription: for Bob's subscription, the values are the stock symbol, XYZ, and the price targets, 45 and 75; for Mary's subscription, the stock symbol is PQS, and the price targets are 80 and 120. To evaluate additional subscriptions that have the same logical form, new rows are needed in the parameters table, but the same query can evaluate them all together.

Because the same fundamental condition can be expressed in many ways using Boolean constructs, SQL-NS attempts to normalize the condition statements before translating them into SQL queries. This maximizes the opportunity for the same query to be used for multiple subscriptions.

Note that the queries and table structures shown here are simplified for ease of explanation. In actual applications, the queries and supporting tables generated by SQL-NS are

somewhat more complex because they are designed to accommodate an unlimited number of parameters per query and support many different data types. However, they are based on the basic principles described here.

Your choice of subscription-evaluation model (developer-defined or user-defined logic) affects the scalability of your applications. Generally, applications that use developer-defined logic are more scalable because subscription grouping is enforced by the developer. With user-defined logic, the extent to which grouping is possible depends on the similarity of the subscriptions entered. In the worst case, if every subscription condition has a different structure, no grouping is possible and the application would have to evaluate one subscription at a time. In practice, it's unlikely that this worst-case scenario would actually play out; in most cases, some degree of grouping is achievable.

Just because you have the option of supporting user-defined logic in your SQL-NS applications does not mean you always should. It's worth questioning whether the flexibility offered by user-defined logic, which requires a sacrifice in scalability, is really needed in your applications. The obvious drawbacks to using developer-defined logic are that all subscriptions must have the same structure, and the only kinds of subscriptions users can enter are those the developer has implemented. But do your users really need to be able to enter arbitrarily complex subscriptions? Does each user need to be able to enter a different kind of subscription? Or can you devise a set of subscription types that cover the overwhelming majority of subscriptions users will want to enter?

Remember that SQL-NS enables you to implement several types of subscriptions in a single application, each with a separate data schema and matching query. Experience has shown that in the vast majority of applications, developers can predict what subscriptions users will want to enter and implement those as predefined subscription types. Often, the lack of flexibility in the subscription-matching logic is not even noticed by the users of these applications.

All the sample applications in Parts I, II, and III of this book (including the sample application in this chapter) use only developer-defined matching logic. The use of user-defined matching logic is covered in Part IV, in Chapter 18, "User-Defined Matching Logic in SQL-NS Applications." At this point, you should focus on learning the fundamental SQL-NS concepts, the majority of which are the same, regardless of which subscription evaluation model you choose.

Programming to the SQL-NS Application Model

In essence, the SQL-NS application model views events and subscriptions as data and uses SQL joins to match them. As a developer building on the SQL-NS platform, you define the following aspects of your application:

- Schemas for the event, subscription, and notification data
- Logic that the application executes to perform matching and maintain state
- Configuration of the SQL-NS execution engine components that run the application

You provide all this information in an XML document called an *Application Definition File (ADF)*. Think of the ADF as the source code for the application: to create a new application, you author a new ADF.

NOTE

SQL-NS provides an XSD schema that defines the XML elements an ADF may contain. As you work through this and subsequent chapters of this book, you'll learn about these XML elements in detail. This chapter highlights specific pieces of the stock notification application's ADF.

When you write an ADF, you usually begin with the elements that define schemas. Each schema is a description of the size and shape of one kind of data. You provide the names of the fields and their data types, much as you would if you were defining a SQL table. The schema for the events specifies the structure of the data your application receives from its event sources. The schema for the subscription data describes the information each subscriber will provide when creating a subscription. The notification schema describes the data content of the notifications your application will deliver.

In addition to the data schemas, the ADF also specifies how events and subscriptions are matched to form notifications. If you choose developer-defined logic as your subscription evaluation model, the matching logic you provide in the ADF consists of actual SQL statements that find the matches. These statements operate on the event and subscription data (as defined by their respective schemas) and produce a row of data for each notification to be sent (the columns in the resultsets correspond to the fields in the notification schemas). If instead you choose user-defined logic as your evaluation model, in the logic sections of the ADF, you provide SQL statements that produce notification data (conforming to the notification schemas) from event and subscription data that has already been determined to match (according to the user-defined conditions).

In the component configuration section of the ADF, you specify how the SQL-NS engine components should run your application. These components perform such functions as gathering event data from event sources, coordinating the execution of the matching logic, and delivering notifications to their destinations. You can specify which components should be used in your application, how they are distributed across various servers, what resources they should use, and when they should run.

When your ADF is complete, you compile it using the SQL-NS compiler. The compiler translates the XML application definition into a set of database objects that will be used to run the application. These include tables for configuration settings, events, subscriptions, and notifications, as well as stored procedures that execute the SQL statements you provided in the ADF. The compiler installs these database objects into a database on your SQL server and populates some of the tables with the configuration information from the ADF. When you run your application, the SQL-NS engine connects to this database, reads the configuration information, and starts its various components; those components then interact with the database (reading and writing data) as they perform their execution functions. This process is illustrated in Figure 3.5.

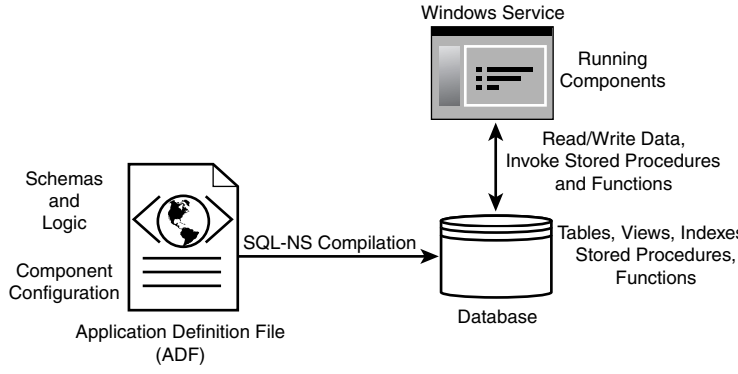

FIGURE 3.5 The ADF is compiled into database structures, and the SQL-NS engine runs the application.

NOTE

Some important parts of a complete, running application—such as the event sources, the subscription management interface, and the delivery systems—are not shown in Figure 3.5. Usually, these other parts of the application also interact with the application database, alongside the SQL-NS engine.

In the remainder of this chapter, we'll create and run the stock notification application. In doing so, we'll look at the data schemas, matching logic, and component configuration, as specified in the application's ADF. Although there are other aspects to the application, we'll focus on these for now because they form the application's core and are the most illustrative of the SQL-NS application model.

NOTE

To work through the steps in this chapter, you need to set up your development environment as described in Chapter 2, "Getting Set Up." If you have not already done so, go through the steps in that chapter before proceeding.

Building the Stock Application's ADF

The ADF for the stock application defines its schemas and logic, as well as its component configuration. The schemas describe the size and shape of events, subscriptions, and notifications; the logic defines how events and subscriptions are matched to produce notifications. The component configuration tells the SQL-NS engine how to actually run the application.

Table 3.1 shows the high-level structure of the ADF (with the application-specific details removed). Think of this as a basic outline for every ADF you will ever write. A root

<Application> element contains various other XML elements that define the parts of the application. This section describes the content that goes in each of these XML elements to create a real application.

NOTE

I've left some rarely used ADF elements out of Table 3.1 for clarity. These include the <History> and <Version> elements, which are intended primarily for developers to track changes to source files and do not affect the operation of the application in any way. These elements are described in the SQL-NS Books Online. Also, some of the elements in Table 3.1 are optional; they will not be present in every ADF. Later chapters cover each element in detail and explain the defaults used when optional elements are omitted.

NOTE

In SQL Server 2005, the SQL-NS Books Online are included as part of the SQL Server Books Online. To open the SQL Server Books Online, go to the Start menu, All Programs, Microsoft SQL Server 2005, Documentation and Tutorials, SQL Server Books Online. The SQL-NS-specific documentation can be found in the "SQL Server Notification Services" topic in the main table of contents.

TABLE 3.1 Basic Structure of the ADF

XML Element	Purpose
`<?xml version="1.0" encoding="utf-8" ?>`	Standard XML header

`<Application`
`xmlns:xsd="http://www.w3.org/2001/XMLSchema"`
`➥xmlns:xsi="http://www.w3.org/2001/`
`➥XMLSchema-instance" xmlns="...">`
	Root element that contains all of the application definition

`<Database>` `</Database>`	Specifies the database in which the application's tables, views, stored procedures, and other objects should be created
`<EventClasses>` `</EventClasses>`	Defines schemas for the events the application will receive
`<SubscriptionClasses>` `</SubscriptionClasses>`	Defines schemas and matching logic for the subscriptions the application will support
`<NotificationClasses>` `</NotificationClasses>`	Defines schemas for the notifications the application will send
`<Providers>` `</Providers>`	Configures the application's event providers

XML Element	Purpose
`<Generator>` `</Generator>`	Configures the generator component that matches events with subscriptions
`<Distributors>` `</Distributors>`	Configures the distributor components that deliver notifications
`<ApplicationExecutionSettings>` `</ApplicationExecutionSettings>`	Contains operational settings that control various aspects of the application's behavior
`</Application>`	Closing tag for the root element

The `<EventClasses>`, `<SubscriptionClasses>`, and `<NotificationClasses>` subelements under `<Application>` contain all the schemas and logic. The other elements provide configuration information used by the SQL-NS compiler and the SQL-NS engine.

The Completed ADF

Before delving into the details of each specific section, take a look at the completed ADF, shown in Listing 3.1. I'm including this here, even before I explain what any of it means, because I find that with almost any program, it helps to get a feel for the code by looking at it from end to end. Just by glancing at it, the ADF will probably make some sense to you, even without further explanation. The following sections describe the important parts in detail.

LISTING 3.1 The Completed Stock ADF

```
<?xml version="1.0" encoding="utf-8" ?>
<Application xmlns:xsd=http://www.w3.org/2001/XMLSchema
➥xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
➥xmlns="http://www.microsoft.com/MicrosoftNotificationServices/
➥ApplicationDefinitionFileSchema">

  <Database>
    <DatabaseName>StockBroker</DatabaseName>
    <SchemaName>StockWatcher</SchemaName>
  </Database>

  <EventClasses>
    <EventClass>
      <EventClassName>StockPriceChange</EventClassName>
      <Schema>
        <Field>
          <FieldName>StockSymbol</FieldName>
          <FieldType>NVARCHAR(10)</FieldType>
```

LISTING 3.1 Continued

```
        <FieldTypeMods>NOT NULL</FieldTypeMods>
      </Field>
      <Field>
        <FieldName>StockPrice</FieldName>
        <FieldType>SMALLMONEY</FieldType>
        <FieldTypeMods>NOT NULL</FieldTypeMods>
      </Field>
    </Schema>
  </EventClass>
</EventClasses>

<SubscriptionClasses>
  <SubscriptionClass>
    <SubscriptionClassName>StockPriceHitsTarget</SubscriptionClassName>
    <Schema>
      <Field>
        <FieldName>StockSymbol</FieldName>
        <FieldType>NVARCHAR(10)</FieldType>
        <FieldTypeMods>NOT NULL</FieldTypeMods>
      </Field>
      <Field>
        <FieldName>StockPriceTarget</FieldName>
        <FieldType>SMALLMONEY</FieldType>
        <FieldTypeMods>NOT NULL</FieldTypeMods>
      </Field>
    </Schema>
    <EventRules>
      <EventRule>
        <RuleName>MatchStockPricesWithTargets</RuleName>
        <Action>
          INSERT INTO [StockWatcher].[StockAlert]
          SELECT subscriptions.SubscriberId,
                 N'DefaultDevice',
                 N'en-US',
                 events.StockSymbol,
                 events.StockPrice,
                 subscriptions.StockPriceTarget
          FROM   [StockWatcher].[StockPriceChange] events
          JOIN   [StockWatcher].[StockPriceHitsTarget] subscriptions
              ON events.StockSymbol = subscriptions.StockSymbol
          WHERE  events.StockPrice &gt;= subscriptions.StockPriceTarget
        </Action>
        <EventClassName>StockPriceChange</EventClassName>
      </EventRule>
```

LISTING 3.1 Continued

```
      </EventRules>
    </SubscriptionClass>
  </SubscriptionClasses>
  <NotificationClasses>
    <NotificationClass>
      <NotificationClassName>StockAlert</NotificationClassName>
      <Schema>
        <Fields>
          <Field>
            <FieldName>StockSymbol</FieldName>
            <FieldType>NVARCHAR(10)</FieldType>
          </Field>
          <Field>
            <FieldName>StockPrice</FieldName>
            <FieldType>SMALLMONEY</FieldType>
          </Field>
          <Field>
            <FieldName>StockPriceTarget</FieldName>
            <FieldType>SMALLMONEY</FieldType>
          </Field>
        </Fields>
      </Schema>
      <ContentFormatter>
        <ClassName>XsltFormatter</ClassName>
        <Arguments>
          <Argument>
            <Name>XsltBaseDirectoryPath</Name>
            <Value>%_ApplicationBaseDirectoryPath_%\XslTransforms</Value>
          </Argument>
          <Argument>
            <Name>XsltFileName</Name>
            <Value>StockAlert.xslt</Value>
          </Argument>
        </Arguments>
      </ContentFormatter>
      <Protocols>
        <Protocol>
          <ProtocolName>File</ProtocolName>
        </Protocol>
      </Protocols>
    </NotificationClass>
  </NotificationClasses>

  <Providers>
    <NonHostedProvider>
```

LISTING 3.1 Continued

```
      <ProviderName>TestEventProvider</ProviderName>
    </NonHostedProvider>
  </Providers>

  <Generator>
    <SystemName>%_NSServer_%</SystemName>
  </Generator>

  <Distributors>
    <Distributor>
      <SystemName>%_NSServer_%</SystemName>
      <QuantumDuration>PT15S</QuantumDuration>
    </Distributor>
  </Distributors>

  <ApplicationExecutionSettings>
    <QuantumDuration>PT15S</QuantumDuration>
  </ApplicationExecutionSettings>

</Application>
```

Before you continue reading, you should open the stock application's source code on your system so that you can browse the ADF as you go. Use the following instructions to open the project in Management Studio and bring up the ADF in the built-in XML editor:

1. Start Management Studio (from the Start menu, choose All Programs, SQL Server 2005, Microsoft SQL Server Management Studio) and connect to your SQL Server.
2. From the File menu, choose Open, Project/Solution.
3. In the File Open dialog box, browse to the `C:\SQL-NS\Samples\StockBroker` directory and select the `StockBroker.ssmssln` solution file.
4. If the Solution Explorer window is not visible, open it by selecting Solution Explorer from the View menu or by pressing Ctrl+Alt+L.
5. In the Solution Explorer, you should see two projects: StockBroker and StockWatcher. Expand the StockWatcher project node and the Miscellaneous folder beneath it.
6. Open the `ApplicationDefinition.xml` file.

The Database Element in the ADF

As described earlier in the "Programming to the SQL-NS Application Model" section (p. 47), the SQL-NS compiler creates database objects (including tables, views, and stored procedures) based on the information in the ADF. You can choose where these database

objects are installed: in the `<Database>` element of the ADF, you can provide a target database name and a schema name. When you compile the ADF, the resulting database objects are installed in the database and schema you specify (see the sidebar titled "User-Schema Separation in SQL Server 2005," p. 55, for an explanation of the term *schema* in this context).

In the case of the stock application's ADF, shown in Listing 3.1, the `<Database>` element specifies a database name as `StockBroker` and the schema name as `StockWatcher`. I chose these names arbitrarily. `StockBroker` seemed like a fitting name for a database related to stockbroker operations and `StockWatcher` aptly describes the purpose of this application. All database objects for this application created by the SQL-NS compiler will be placed in the `StockWatcher` schema in the `StockBroker` database. In the later sections of this chapter, you'll see these database objects referred to by their schema-qualified names.

NOTE

If the database and schema specified in the `<Database>` element don't exist at compilation time (as will likely be the case on your system), the SQL-NS compiler will create them.

USER-SCHEMA SEPARATION IN SQL SERVER 2005

"Schema" is one of the most overloaded words in database terminology. It generally refers to a description of the structure and organization of some kind of data. However, in SQL Server 2000 and SQL Server 2005, *schema* also refers to a namespace for a group of related database objects. This sidebar explains this usage of the term. Wherever the word schema is used in the remainder of this book, I explicitly call out the intended meaning, if it isn't obvious from the context.

In SQL Server 2000, schemas were used to group objects in a database created by a single user. Each user had an associated schema (the schema name was the same as the username). A user's schema isolated the database objects she created from those created by other users. For example, if user Jane created a table called Products, the table would, by default, go into the Jane schema. Its *schema-qualified name* would be "Jane.Products". This would allow another user—for example, Joe—to also create a Products table, without resulting in a name collision (Joe's table would be called "Joe.Products").

In SQL Server 2005, schemas are separated from users. You can create schemas independently of users, simply to group related database objects. For example, in a given database, you can create a schema called Inventory, and then create a table within this schema called Products. The schema-qualified name for this table would then be "Inventory.Products". You can also create other schemas and place Products tables in those schemas without name collisions. For example, you can create a Manufacturing schema, with a "Manufacturing.Products" table.

Schema-qualified names can be used in any place nonqualified names are normally used. For example, to write a query that refers to tables in specific schemas, you would include the schema-qualified table names in the `FROM` clause.

Schemas and Logic

This section describes the ADF syntax used to define the event, subscription, and notification data schemas (here I'm referring to the usual meaning of schemas—the definition of the data's structure). This section also shows how the matching logic, expressed as a SQL join, is specified in the ADF.

Event Schemas

For each type of event that the application receives (there can be more than one), you must declare an event class in the `<EventClasses>` element of the ADF.

The stock application uses only a single type of event: a change in the trading price of a stock. Within the `<EventClasses>` element of the ADF, you declare an `<EventClass>` for this type of event, as shown in Listing 3.2.

LISTING 3.2 Declaration of the `StockPriceChange` Event Class

```
<EventClass>
  <EventClassName>StockPriceChange</EventClassName>
  <Schema>
    <Field>
      <FieldName>StockSymbol</FieldName>
      <FieldType>NVARCHAR(10)</FieldType>
      <FieldTypeMods>NOT NULL</FieldTypeMods>
    </Field>
    <Field>
      <FieldName>StockPrice</FieldName>
      <FieldType>SMALLMONEY</FieldType>
      <FieldTypeMods>NOT NULL</FieldTypeMods>
    </Field>
  </Schema>
</EventClass>
```

The declaration provides a name, StockPriceChange, for the event class and a schema for the event data. The schema declaration syntax is basically an XML version of a SQL CREATE TABLE statement. It declares a set of fields specifying a data type and, optionally, a type modifier (such as not null) for each.

When building the application's database, the SQL-NS compiler processes this event class declaration and constructs a table that will ultimately store the events. It creates a column for each of the fields declared in the event class and adds some other columns used for internal tracking.

The SQL-NS compiler also builds a view over the events table. The columns in the view match the fields declared in the event class exactly—the extra columns created in the events table for SQL-NS internal tracking are not present in the view. This view becomes the primary means by which events are manipulated in the application. To submit event data for processing, you insert rows into the view. Triggers on the view (also created by

the SQL-NS compiler) handle inserting the appropriate values into the internal columns in the events table. When matching against subscriptions, event data is read from the view.

Subscription Schemas

Just as you have to declare an event class for each type of event the application receives, you also have to declare a subscription class for each type of subscription that the application supports. This simple stock application has only one type of subscription: a request to be notified when a stock hits a particular price target. In the completed ADF, this subscription class declaration goes within the `<SubscriptionClasses>` element. Listing 3.3 shows the subscription class declaration.

LISTING 3.3 Declaration of the `StockPriceHitsTarget` Subscription Class

```
<SubscriptionClass>
  <SubscriptionClassName>StockPriceHitsTarget</SubscriptionClassName>
  <Schema>
    <Field>
      <FieldName>StockSymbol</FieldName>
      <FieldType>NVARCHAR(10)</FieldType>
      <FieldTypeMods>NOT NULL</FieldTypeMods>
    </Field>
    <Field>
      <FieldName>StockPriceTarget</FieldName>
      <FieldType>SMALLMONEY</FieldType>
      <FieldTypeMods>NOT NULL</FieldTypeMods>
    </Field>
  </Schema>
  <EventRules>
    ...
  </EventRules>
</SubscriptionClass>
```

The subscription class declaration has three subelements: a name, a schema, and a set of event rules (content is not shown in the fragment in Listing 3.3).

The schema subelement defines the size and shape of the subscription data. In this application, we're using developer-defined matching logic and constraining subscriptions to the form

"Notify me when the price of stock *S* goes above price target *T*."

The subscription data for each subscription then just consists of values for *S* and *T*. In the subscription class schema, we've given these fields the more descriptive names `StockSymbol` and `StockPriceTarget` and provided their data types and type modifiers.

Just as it does with the event class schema declaration, the SQL-NS compiler constructs a table for subscription data from the subscription class schema declaration. Like the events

table, the subscriptions table contains a column for each field declared and some extra columns used internally by SQL-NS. Also, the SQL-NS compiler constructs a view over the subscriptions table, with columns matching the declared subscription class fields. When you need to add subscriptions of this subscription class, you insert rows into the subscriptions view. When you define the logic that matches subscriptions against events, you access subscription data by selecting from the subscriptions view.

The matching logic for the subscription class is declared in the `<EventRules>` subelement. This is discussed in the "Matching Logic" section (p. 59).

Notification Schemas

The `<EventClasses>` and `<SubscriptionClasses>` elements define the schemas for the application's events and subscriptions. When the matching logic is applied, the result is a set of notification data that represents the notifications to be sent to subscribers. This notification data has a schema as well and must be declared in the `<NotificationClasses>` element of the ADF. This section can contain definitions for several types of notifications, but because this stock application example just sends one type of notification, there is just a single `<NotificationClass>` declaration, shown in Listing 3.4.

LISTING 3.4 Declaration of the `StockAlert` Notification Class

```
<NotificationClass>
  <NotificationClassName>StockAlert</NotificationClassName>
  <Schema>
    <Fields>
      <Field>
        <FieldName>StockSymbol</FieldName>
        <FieldType>NVARCHAR(10)</FieldType>
      </Field>
      <Field>
        <FieldName>StockPrice</FieldName>
        <FieldType>SMALLMONEY</FieldType>
      </Field>
      <Field>
        <FieldName>StockPriceTarget</FieldName>
        <FieldType>SMALLMONEY</FieldType>
      </Field>
    </Fields>
  </Schema>
  <ContentFormatter>
    ...
  </ContentFormatter>
  <Protocols>
    ...
  </Protocols>
</NotificationClass>
```

Much like in the event and subscription class declarations, the <Schema> element provides the names and data types of the notification fields. The <ContentFormatter> section describes how the notification data is formatted for receipt by the subscriber, and the <Protocols> section declares which delivery protocols can be used to actually send the notifications. The <ContentFormatter> and <Protocols> elements are discussed more thoroughly in Chapters 5, 9, and 10.

The StockAlert notification schema has three fields: the stock symbol, stock price, and stock price target. Each row of notification data produced by the matching join contains a value for each of these fields. The stock price field contains the current price of the stock (as indicated by the stock event), and the stock price target field contains the price target specified in the subscription. From this notification data, the application can synthesize formatted stock alert messages for delivery to the subscribers, such as

"XYZ is now trading at: $55.55. This is greater than or equal to the target price of $50.00."

As you might expect, the SQL-NS compiler constructs a table for the notification data (based on the declared schema) and a view over this table. The output of the matching logic query is inserted into the notifications view to queue notifications for delivery.

Matching Logic

The stock application's matching logic is specified in the <EventRules> section of the subscription class. This section contains one or more SQL statements that SQL-NS executes when events arrive. Each of these SQL statements is called a *rule* and is declared in an <EventRule> element. Each rule specifies a name, an action (the SQL query to execute), and the name of the event class that triggers it.

Listing 3.5 shows the <EventRules> element of the StockPriceHitsTarget subscription class. It declares a single rule, MatchStockPricesWithTargets, that simply matches incoming events with StockPriceHitsTarget subscriptions. The <EventClassName> element of the rule declaration specifies the name of the triggering event class—in this case, StockPriceChange. This instructs the SQL-NS execution engine to fire this event rule whenever events of the StockPriceChange event class arrive.

LISTING 3.5 Event Rules Declaration Within the StockPriceHitsTarget Subscription Class

```
<EventRules>
  <EventRule>
    <RuleName>MatchStockPricesWithTargets</RuleName>
    <Action>
      ...
    </Action>
    <EventClassName>StockPriceChange</EventClassName>
  </EventRule>
</EventRules>
```

The matching logic query is specified in the rule's <Action> element. Listing 3.6 shows the contents of the <Action> element from the stock application's ADF.

LISTING 3.6 The SQL Matching Logic from the Event Rule's <Action> Element

```
INSERT INTO [StockWatcher].[StockAlert]
SELECT subscriptions.SubscriberId,
       N'DefaultDevice',
       N'en-US',
       events.StockSymbol,
       events.StockPrice,
       subscriptions.StockPriceTarget
FROM   [StockWatcher].[StockPriceChange] events
JOIN   [StockWatcher].[StockPriceHitsTarget] subscriptions
    ON events.StockSymbol = subscriptions.StockSymbol
WHERE  events.StockPrice &gt;= subscriptions.StockPriceTarget
```

This logic is just a SQL join that produces a set of rows that becomes notification data.

The first thing to look at in this query is the FROM clause. It selects data from the events view joined with the subscriptions view. Notice that the names of these views are the names of the event class and the subscription class, respectively (with the StockWatcher schema qualifiers). The aliases, events and subscriptions, have been assigned to these views for clarity (these aliases are not mandated by SQL-NS).

The event and subscription views, StockPriceChange and StockPriceHitsTarget, can be thought of as the SQL-NS platform's public interface to the event and subscription data collected. The platform guarantees that these views will contain only the event and subscription data against which the rule should operate, at the time the rule is invoked. In other words, the events view will contain only the events just submitted that have triggered the rule firing, and the subscriptions view will contain only the active subscriptions. (In this simple application, all subscriptions are active, but you'll see in Chapter 6, "Completing the Application Prototype: Scheduled Subscriptions and Application State," and Chapter 7, "The SQL-NS Subscription Management API," that subscriptions can be disabled or scheduled to fire only at certain times.)

Given the guarantee that the views always contain only the relevant data when the rule is invoked, the matching query in the ADF doesn't require any custom logic to determine which portions of the data are in scope. This is a great benefit of using SQL-NS: the application logic can be written at a high level of abstraction, leaving the platform to take care of details such as managing the data against which the logic executes.

NOTE

The scoping of data in the event and subscription views happens because the view definitions refer to internal control tables (set up by SQL-NS at runtime) to select only the relevant rows from the underlying event and subscription tables.

Going back to the query in Listing 3.6, the `WHERE` clause defines a filter that selects only the rows in which the stock price in the event is greater than or equal to the stock price target in the subscription. Note that the XML escape sequence `>` is used in place of the > character because this statement appears within an XML document. Use of the > character directly would prevent the document from being well-formed XML.

The join defined in the `FROM` clause, along with the filter defined in the `WHERE` clause, implement the matching criteria; the logic specified in the query defines what it means for an event to match a subscription. (The stock symbols must be the same, and the stock price greater than or equal to the price target.) This is an example of developer-defined logic; users cannot change this definition of what a match means.

The results obtained when the matching query is executed determine what notifications will ultimately be delivered. Each row in the resultset represents one notification. Notice that the results of the `SELECT` query are inserted into the notification class view (which, as you might expect, has the same name as the notification class, `StockAlert`, defined in Listing 3.4). Inserting rows into a notification class view causes notifications of that notification class to be generated. As is the case for event and subscription classes, the view is the SQL-NS platform's interface to the notification class. Rows inserted into the notification class view will eventually be picked up by the SQL-NS distributors for final formatting and delivery.

The notification class view has a column for each field declared in the notification class. Given the definition of the notification class schema in Listing 3.4, the `StockAlert` view has columns for `StockSymbol`, `StockPrice`, and `StockPriceTarget`. SQL-NS also adds three other columns to every notification class view:

- `SubscriberId`—Specifies the ID of the subscriber to receive the notification.
- `DeviceName`—Specifies the name of the subscriber's device to which the notification should be sent. (Subscriber devices are discussed in more detail in Chapter 7 and Chapter 10, "Delivery Protocols.")
- `SubscriberLocale`—Specifies the locale for which the notification data should be formatted.

The `SELECT` clause in Listing 3.6 provides a value in the resultset for each of the columns in the notification class view. The subscriber ID is obtained from the subscription, the device name and locale are constants, two of the notification fields come from data in the event, and the third comes from data in the subscription.

Component Configuration and the Phases of Processing

Previous sections examined the event, the subscription, and the notification schemas and the associated rule logic. This section examines the component configuration elements of the ADF. Specifically, these include the `<Providers>`, `<Generator>`, `<Distributors>`, and `<ApplicationExecutionSettings>` elements.

Each of the components configured in the ADF plays a specific role in the functioning of the application. Before looking at these components and how they're configured, it's important to understand the processing that happens inside a notification application.

SQL-NS separates the functions of a notification application into separate processing phases:

- Event collection—Gathering events and submitting them to the application
- Subscription management—Creation, deletion, and alteration of subscriptions
- Generation—Matching events with subscriptions
- Distribution—Routing notifications to delivery systems

Figure 3.6 shows these phases.

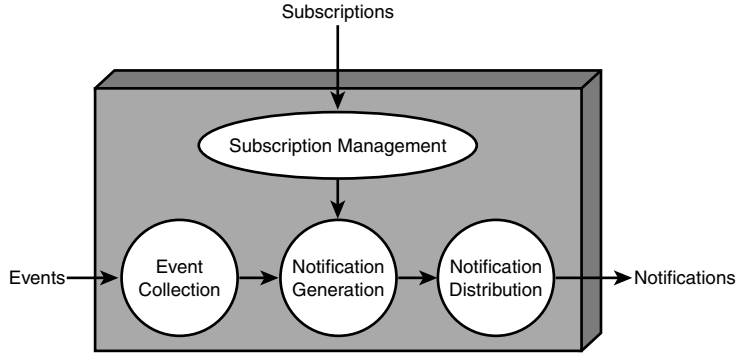

FIGURE 3.6 Phases of processing in notification applications.

Each phase is considered independent and is handled by a separate component in the execution engine. The SQL-NS engine may run all these phases concurrently. While one batch of events is being collected, another may be being matched with subscriptions, and an older batch of notifications may be being distributed. The following subsections describe these phases of processing and the components that execute them.

Event Collection

Event providers are components that collect events and submit them to notification applications. Event providers can gather event data from the outside world proactively, or act as sinks to which external event sources push information. For example, an event provider could be a component that constantly polls an external data source, or it could be a web service that receives data passed to it by external callers.

An application can have several event providers, each potentially submitting events from a different event source. Event providers can be components that run within the SQL-NS engine, or they can be standalone programs that submit event data to your application. In the `<Providers>` element of the ADF, you declare which event providers your application will use and configure the options that control their operation.

SQL-NS provides several built-in event providers that can be used in an application without your having to write any code. You can also build your own event provider for your application that talks to a custom event source. Chapter 8, "Event Providers," provides details on all the built-in event providers, as well as the process of building a custom event provider for your application.

Listing 3.7 shows the `<Providers>` element from the stock application's ADF.

LISTING 3.7 The Event Provider Configuration in the ADF

```
<Providers>
  <NonHostedProvider>
    <ProviderName>TestEventProvider</ProviderName>
  </NonHostedProvider>
</Providers>
```

In this application, rather than using a real event source, we're just going to submit some test events manually. Therefore, the ADF declares a single *nonhosted event provider*. The term "nonhosted" just means that the event provider is not hosted within the SQL-NS engine; the declaration in Listing 3.7 indicates that events will enter the application from an external program (in fact, we'll use a simple T-SQL script to submit events). Because there's only one event source in this application example, the event provider name is arbitrary. I've chosen the name `TestEventProvider` to signify that this is a placeholder used for testing. In a real application, which might have several real hosted and nonhosted event providers, event provider names can be useful to associate events with the providers that submitted them.

NOTE

Chapter 8 offers more details on both hosted and nonhosted event providers and describes when it's appropriate to use each type.

Subscription Management

Most notification applications provide users with a visual interface by which they can manipulate and manage their subscriptions. The form of this interface varies, depending on the nature of the pub-sub system. A stock quote application such as the one in this chapter might provide a website that a user can visit to enter a subscription. A line-of-business application might provide a way to subscribe for notifications in its standard Windows user interface.

Whatever the form of the external interface, subscription data must be ultimately transferred to the tables in the application's database. The implementation of the subscription management system can either insert rows into the subscription class views directly or use an API provided by SQL-NS to enter subscriptions. The subscription management API is covered in detail in Chapter 7.

There is no subscription management configuration in the ADF. The ADF contains declarations of subscription classes, but the systems by which subscriptions of those subscription classes are created and submitted to the application are treated as standalone entities that are not configured in the ADF. For the purposes of testing the application example in this chapter, we'll just enter subscription data directly into the views using a T-SQL script.

Generation

Generation is the phase during which events are matched with subscriptions to produce notifications. The generation component is provided by SQL-NS and uses either developer-defined or user-defined logic to determine whether a particular event matches a subscription.

SQL-NS exposes a variety of configuration options for controlling notification generation. Some of these options specify, for example, how and when matching logic is applied. These are configured in the ADF in the `<Generator>` and `<ApplicationExecutionSettings>` elements. Listing 3.8 shows the generator configuration for the stock application.

LISTING 3.8 The Generator Configuration in the ADF

```
<Generator>
  <SystemName>%_NSServer_%</SystemName>
</Generator>
  ...
<ApplicationExecutionSettings>
  <QuantumDuration>PT15S</QuantumDuration>
</ApplicationExecutionSettings>
```

The `<Generator>` element specifies the system name, which tells the SQL-NS engine on which machine the generator should run. The `<ApplicationExecutionSettings>` element specifies a *quantum duration*, which controls how often the generator looks for new events to process. In this case, the quantum duration is set to 15 seconds. Chapters 11, "Debugging Notification Generation," and 12, "Performance Tuning," describe in more detail these and other options you can use to fine-tune generator operation.

Distribution

The distribution phase handles the formatting of notification data for a variety of delivery devices (email, cell phones, pagers, and so on) and the routing of the formatted notifications to delivery systems that gets them to their final destinations.

An application can have one or more distributors, possibly running on different servers. These are configured in the ADF's `<Distributors>` element. Listing 3.9 shows the `<Distributors>` element from the stock application's ADF.

LISTING 3.9 The Distributor Configuration in the ADF

```
<Distributors>
  <Distributor>
    <SystemName>%_NSServer_%</SystemName>
    <QuantumDuration>PT15S</QuantumDuration>
  </Distributor>
</Distributors>
```

The stock application uses only one distributor, and this is declared in the single `<Distributor>` element. The configuration in Listing 3.9 specifies the distributor system name (the computer on which the distributor should run) and the distributor quantum duration (which controls how often the distributor polls for new batches of notifications to format and deliver—15 seconds, in this case). Chapter 12 covers these and other distributor configuration options in greater detail.

CAUTION

If you are using the Standard Edition of SQL-NS, the event providers, generator, and distributor in your application must run on the same machine. This means that the `<SystemName>` element must have the same value in all the event provider, generator, and distributor declarations in your ADF. If you specify different system name values with SQL-NS Standard Edition, you get an error when compiling your application.

If you are using the Enterprise Edition of SQL-NS, you may configure the various components to run on different machines by specifying different values for the `<SystemName>` elements. This allows your application to scale out for better performance. Chapter 13, "Deploying a SQL-NS Instance," covers the additional configuration steps (beyond specifying system names in the ADF) required to enable a scale-out deployment.

Specifying Other Parts of the Stock Application

The ADF defines the core of the application, but several other pieces are required to make it run. This section briefly describes those pieces as they pertain to the stock application.

The Instance

SQL-NS has the concept of instances, in much the same way that SQL Server does. A SQL-NS instance is a single, named configuration of SQL-NS that can host one or more applications. Each instance is an independent entity that can be started, stopped, and configured on its own. In SQL-NS, an instance is defined by an Instance Configuration File (ICF). The ICF defines the list of applications in the instance (here, just one—the `StockWatcher` application) and the set of delivery channels that applications in the instance can use to send notifications.

This sample's ICF creates a Notification Services instance called StockBroker. I've included the text of the ICF in Listing 3.10, but I will not go into an explanation of it here because most of the details are not relevant at this point. Chapter 4, "Working with SQL-NS

Instances," covers the instance concepts and all the elements of the ICF in detail. If you have the Management Studio project for this sample open, you'll find the ICF under the Miscellaneous node in the StockBroker project—the file name is `InstanceConfiguration.xml`.

LISTING 3.10 The ICF That Defines the SQL-NS Instance That Hosts the Stock Application

```
<?xml version="1.0" encoding="utf-8"?>
<NotificationServicesInstance xmlns:xsd=http://www.w3.org/2001/XMLSchema
➥xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
➥xmlns="http://www.microsoft.com/MicrosoftNotificationServices/
➥ConfigurationFileSchema">

  <InstanceName>StockBroker</InstanceName>

  <ParameterDefaults>
    <Parameter>
      <Name>_SQLServer_</Name>
      <Value>... Your SQL Server Name ...</Value>
    </Parameter>
    <Parameter>
      <Name>_NSServer_</Name>
      <Value>%COMPUTERNAME%</Value>
    </Parameter>
    <Parameter>
      <Name>_InstanceBaseDirectoryPath_</Name>
      <Value>C:\SQL-NS\Samples\StockBroker</Value>
    </Parameter>
  </ParameterDefaults>

  <SqlServerSystem>%_SQLServer_%</SqlServerSystem>

  <Database>
    <DatabaseName>StockBroker</DatabaseName>
    <SchemaName>NSInstance</SchemaName>
  </Database>

  <Applications>
    <Application>
      <ApplicationName>StockWatcher</ApplicationName>
      <BaseDirectoryPath>%_InstanceBaseDirectoryPath_%\StockWatcher
➥</BaseDirectoryPath>
      <ApplicationDefinitionFilePath>ApplicationDefinition.xml
➥</ApplicationDefinitionFilePath>
      <Parameters>
```

LISTING 3.10 Continued

```
        <Parameter>
          <Name>_NSServer_</Name>
          <Value>%_NSServer_%</Value>
        </Parameter>
        <Parameter>
          <Name>_ApplicationBaseDirectoryPath_</Name>
          <Value>%_InstanceBaseDirectoryPath_%\StockWatcher</Value>
        </Parameter>
      </Parameters>
    </Application>
  </Applications>

  <DeliveryChannels>
    <DeliveryChannel>
      <DeliveryChannelName>FileChannel</DeliveryChannelName>
      <ProtocolName>File</ProtocolName>
      <Arguments>
        <Argument>
          <Name>FileName</Name>
          <Value>%_InstanceBaseDirectoryPath_%\FileNotifications.txt</Value>
        </Argument>
      </Arguments>
    </DeliveryChannel>
  </DeliveryChannels>

</NotificationServicesInstance>
```

As it does with the ADF, the SQL-NS compiler creates a set of database objects for the instance from the information provided in the ICF. The `<Database>` element in the ICF specifies where these objects go. In this case, you can see that they are deployed into the same `StockBroker` database as the application objects, but isolated in a different schema called `NSInstance`.

Notice that the ICF contains a reference to the ADF for each declared application. In Listing 3.10, the `<Application>` element that declares the `StockWatcher` application includes an `<ApplicationDefinitionFilePath>` element that provides the path to the ADF. At compile time, you pass just the ICF to the SQL-NS compiler. Based on the `<ApplicationDefinitionFilePath>` elements in the ICF, the compiler locates, loads, and compiles each application's ADF.

Entering Event and Subscription Data

If this were a real application, it would use an event provider that would read and submit real stock data from one of many stock market data publishers. It would also have a subscription management interface by which users could enter subscriptions. Because the

focus of this chapter is the core SQL-NS concepts, not the mechanics of reading from stock market data sources or implementing user interfaces for subscription management, we'll use a simple test script to submit event and subscription data. The data submitted through this script will allow us to observe how the application and the SQL-NS platform work. Chapters 7 and 8 cover the building of subscription management interfaces and event providers in detail.

The test script that submits event and subscription data is included in the StockBroker Management Studio project. To open the script, expand the Queries node under the StockBroker project in Solution Explorer and select the `Test.sql` file. Do not run this file yet—the database objects it relies on don't exist yet because we haven't compiled the SQL-NS ICF and ADF. Just open the file to look at its contents for now.

The script first creates subscriber and subscriber device records for four test subscribers. The records are created by inserting rows into subscriber and subscriber device views in the instance schema in the `StockBroker` database. It then enters a subscription for each subscriber by inserting rows into the subscription class view in the `StockWatcher` application schema. Finally, it inserts a set of test events into the event class view. After compiling the application, we'll run this script to test it.

Seeing the Final Notifications

After raw notification data is generated, it gets formatted and then delivered. In this sample, we'll use the built-in `XsltFormatter` content formatter provided by SQL-NS that turns the raw notification into a readable message using an XSL transform. Although the XSLT formatter is used in the examples in the next several chapters, I'll defer explaining the details of how it works until Chapter 9, "Content Formatters."

On the delivery side, this sample application uses another SQL-NS built-in component: the File Delivery Protocol. This component "delivers" the notifications to a file. You can then check the output of the application simply by opening this file. The subscriber device records created by the test script you examined in the previous section tell the SQL-NS distributor to route the formatted notifications for the test subscribers to the output file. In a real application, you'd probably use several delivery protocols that could send the notification messages via email or as text messages to cell phones, for example. Chapter 10 describes all the built-in SQL-NS delivery protocols and shows how you can build your own custom delivery protocol.

Running the Stock Application

In this section, we'll compile, register, and enable the application; start its engine running; and then feed it sample data. We'll submit events and subscriptions as described earlier and then observe the notification data written to the output file.

Assuming that your development environment is set up as described in Chapter 2, you can complete the steps described here to get the application going. In this chapter, I've just listed the steps without explaining the purpose of each one in detail. Chapter 4 goes into the details of what these steps actually do.

1. If you have not already done so, open the `C:\SQL-NS\Samples\StockBroker\StockBroker.ssmssln` solution in Management Studio.
2. If the Object Explorer pane is not visible, open it by selecting Object Explorer from the View menu, or by pressing F8.
3. In the Object Explorer, connect to the SQL Server you plan to use for SQL-NS development. To do this, click the Connect button, choose Database Engine from the drop-down list, enter the connection information in the dialog box that appears, and finally, click Connect. If you're using SQL Server Authentication, make sure you connect using the credentials of your development account.
4. In the Object Explorer tree, find the Notification Services folder. It should be empty, unless any SQL-NS instances already exist on the SQL Server.
5. Right-click the Notification Services folder and select New Notification Services Instance.
6. In the New Notification Services Instance dialog box, click the Browse button to browse for the stock instance's ICF. In the File Open dialog box, navigate to `C:\SQL-NS\Samples\StockBroker` and select the `InstanceConfiguration.xml` file.

CAUTION

If the `C:\SQL-NS\Samples\StockBroker\InstanceConfiguration.xml` file appears to be missing on your system, you probably have not completed all the setup steps given in Chapter 2. If this is the case, return to Chapter 2 and make sure you complete the steps in the "Customizing the Source Files for Your Environment" section (p. 39).

7. The Parameters grid in the New Notification Services Instance dialog box will be populated with the parameters from the ICF. You should not have to change any of these values.
8. Check the box labeled Enable Instance After It Is Created, below the Parameters grid.
9. Click the OK button to begin compiling the instance and its associated application. A progress dialog box will appear, listing the various steps being performed. When all steps are complete, you should see the status reported as `Success`. When this happens, click Close to dismiss the progress dialog box.
10. Now you should see a new item in the Notification Services folder in the Object Explorer tree: an instance named `StockBroker`.
11. Right-click this new instance and select Tasks, and then select Register from the context menus. The Register Instance dialog box opens.
12. Check the box labeled Create Windows Service.
13. Beneath the Service Logon label, enter the name and password for the Windows

user account you created in Chapter 2 to run the SQL-NS service. If your service user account is a domain account, don't forget to qualify the account name with the domain name.

14. If you're using SQL Server Authentication, select the option button labeled SQL Server Authentication beneath the Authentication label. Enter the SQL login name and password for the test account you created in Chapter 2. If you're using Windows Authentication, leave the Windows Authentication option button selected.

15. Click OK to begin registering the instance. A progress dialog box listing the various registration operations will appear. When registration completes successfully, click the Close button to dismiss the progress dialog box.

16. In the Solution Explorer, expand the Queries folder under the StockBroker project and open the `SQLPermissions.sql` file. Execute the code in this file by pressing F5. This grants the SQL-NS service permission to access the application's database.

17. Right-click the `StockBroker` instance in the Object Explorer and select Start from the context menu. Click Yes in the confirmation dialog box to proceed with starting the SQL-NS service for the instance.

At this point, the application should be running, and we can start submitting subscriptions and events. If your service does not start correctly, refer to Chapter 15, "Troubleshooting SQL-NS Applications," for suggestions on how to diagnose the problem.

To add subscribers, subscriptions, and events, we'll use the T-SQL test script you examined earlier:

1. Open the `Test.sql` file in Management Studio. The file is located under the Queries folder in the StockBroker project.

2. Before you run `Test.sql`, open the `FileNotifications.txt` file from the Miscellaneous folder under the StockBroker project in the Solution Explorer. This is the output file to which the notifications will be written when events are matched against subscriptions. When you first open it, the file should be blank.

3. Go back to `Test.sql` and execute its contents by pressing F5.

CAUTION

If the `Test.sql` script reports error messages saying that it failed to add subscribers because subscription management is disabled for the instance, you probably forgot to check the Enable Instance After It Is Created check box in step 8 of the earlier instructions. If you see these error messages, right-click the StockBroker instance in the Object Explorer and select Enable from the context menu. Click Yes in the confirmation dialog box and then rerun `Test.sql`.

After `Test.sql` completes, the SQL-NS engine begins processing the events and subscriptions by executing the matching logic query defined in the ADF. The distributor in the SQL-NS engine picks up the rows of data that result from the matching query, formats them, and then writes the formatted messages out to the notifications file. Within about a minute, Management Studio should pop up a dialog box telling you that the `FileNotifications.txt` file has changed and asking you if you want to reload it. Click Yes to reload the file and you will see the contents of the notifications that were generated.

The particular event and subscription data submitted by the `Test.sql` script was chosen to generate some matches. If you go back and look at the data values in `Test.sql`, you'll easily see which events and subscriptions satisfy the matching criteria defined by the query in the ADF. The contents of the `FileNotifications.txt` file should reflect these matches.

Feel free to edit `Test.sql` and enter some new events and subscriptions with different values. If you do this, be certain that you don't rerun the statements that enter subscriber and subscriber device records: SQL-NS does not allow duplicate subscribers or subscriber devices, so these statements should be run only once. After you submit new events and subscriptions, any matches that occur will result in new notification messages written to `FileNotifications.txt`.

Inside the Running Application

When the SQL-NS compiler processed the ADF for the stock application, it created database objects from the information in it. When we registered the instance, SQL-NS installed a Windows Service to host its execution engine. These two pieces, the database objects and the Windows Service, are the key components of the running application. This section examines both to illustrate how the application really works.

The Database

Connect the Object Explorer in Management Studio to your SQL Server and expand the Databases node. You should see a new database called `StockBroker` (recall that this is the database name specified in the `<Database>` elements in the ICF and ADF). The SQL-NS compiler created this database when it compiled the ICF and placed database objects associated with the instance and application in it.

The instance database objects are created in a schema called `NSInstance` (this was the name specified in the `<SchemaName>` element in the ICF). If you expand the `StockBroker` database and look at the list of tables, views, and stored procedures, you'll see several objects prefaced with the `NSInstance` schema name. For now, we'll ignore these instance database objects and focus on the application database objects instead. We'll revisit the instance database objects in Chapter 4.

In the list of tables, views, and stored procedures in the `StockBroker` database, many objects are qualified with the `StockWatcher` schema name. These are all the application database objects the compiler created for our `StockWatcher` application. Many of these are

for internal use by the SQL-NS engine, so we won't explore them here. But I do want to show you a few of them to give you an idea of what the compiler did with the information in the ADF.

Look for a table called `StockWatcher.NSStockPriceChangeEvents`. If you look at the columns in this table, you'll see that all the fields from the `StockPriceChange` event class declaration have become columns in this table. SQL-NS has also added its own extra columns that it uses for internal tracking. This table is used to store events and was created by the compiler from the event class definition. Similarly, you'll see a table called `StockWatcher.NSStockPriceHitsTargetSubscriptions` that matches the schema of the `StockPriceHitsTarget` subscription class and is used to store subscriptions of that class. As you might expect, there is also a `StockWatcher.NSStockAlertNotifications` table that stores `StockAlert` notification data.

The Views node under the database in the Object Explorer shows the list of views the compiler created. In this list, you'll find all the views used to insert data in the `Test.sql` script we ran earlier.

Also, look at the `StockWatcher.NSFireDeveloperDefined1` stored procedure. In Object Explorer, stored procedures are located under the Programmability node beneath the database, in the Stored Procedures subnode. To see the definition of the `StockWatcher.NSFireDeveloperDefined1` stored procedure, right-click it, select Script Stored Procedure As, CREATE To, and then New Query Editor Window in the series of context menus. As the definition shows, `StockWatcher.NSFireDeveloperDefined1` wraps the text of the `<Action>` element in the match rule declaration in the ADF. Look for a comment that says `/** APPLICATION DEFINED RULE STARTS HERE **/`. Immediately after this comment, you'll see the rule text we supplied in the ADF. When the SQL-NS engine needs to invoke this matching logic, it executes this stored procedure.

Spend some time looking at other objects in the database to get a feel for what's there. Don't be alarmed by the number of objects you see—you'll only ever have to deal with a small number of these as you build SQL-NS applications. The rest are there to support the rich features that SQL-NS provides.

The Windows Service

In addition to the tables and stored procedures just examined, the SQL-NS compiler also stores the component configuration information from the ADF in the database. On startup, the Windows service connects to the database and reads this configuration. This includes the number and type of event providers, as well as the configuration of the generator and distributors. Using this configuration information, the service instantiates the appropriate execution components and starts them running.

Our application had a single generator, configured with a quantum duration of 15 seconds. On startup, the Windows service instantiated a generator component that checks the database for new batches of events every 15 seconds. When our test script wrote events to the database, the generator processed them by executing the stored procedures that contain our rule logic. This resulted in notification data stored in the notification tables.

The single distributor we configured in our ADF was also represented by a running component in the Windows service, instantiated on startup. This component periodically (again, every 15 seconds, as specified by the distributor's quantum duration) checks the database for new notifications to format and deliver. When the evaluation of the matching logic by the generator caused the notifications to be written to the notifications table, the distributor read them, passed them through the content formatter we specified, and then wrote them to the output file using the file delivery protocol.

Figure 3.7 shows the Windows service and its interactions with the database.

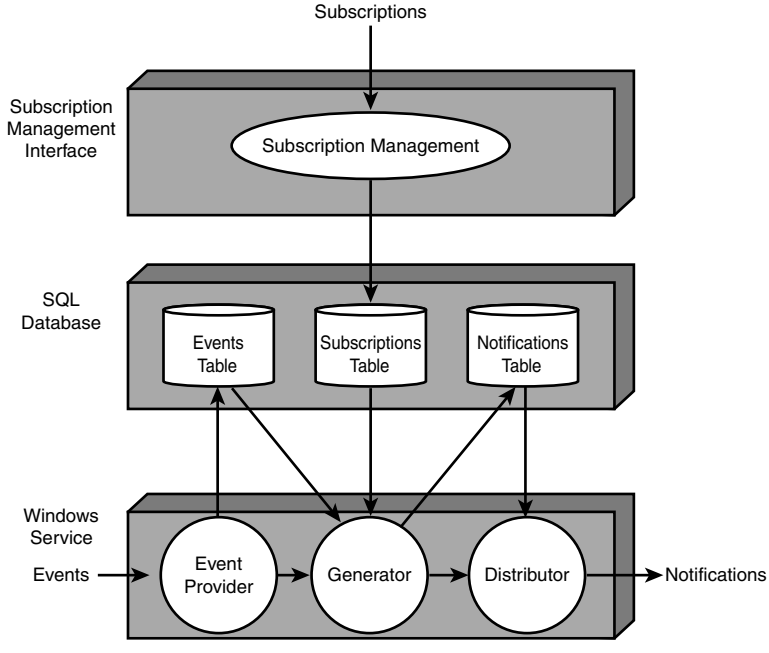

FIGURE 3.7 The running Windows service interacting with the database.

What Has the SQL-NS Platform Provided?

In its final form, the notification application is a database application coordinated by a Windows service. To build the application, we had to write a relatively small amount of code. In fact, all we provided were the schemas for events, subscriptions, and notifications; the matching logic; and component configuration. From this, SQL-NS constructed the database application, including the tables, views, and stored procedures.

In the previous section, we examined the parts of the database that resulted from the information specified in the ADF. We did not look into the multitude of other supporting tables, views, stored procedures, and functions required by the running application. These include code that maintains state about the application and enables it to recover gracefully after an unplanned shutdown, support for garbage collection of old event and

notification data that has been completely processed, and several reporting and debugging utilities that can provide insight into what the running application is actually doing. All these were created for us, and we did not have to understand how they are implemented.

Many other benefits of building this application on SQL-NS have not been visible because we've run the application on such a small scale. But in fact, the application as it stands will support large volumes of events and subscriptions. (The exact number depends on the hardware you're using, but even on a modest developer system, it's possible to run applications with tens of thousands of events and subscriptions.)

Apart from the supported data volume, another aspect of the application that has not been visible because of the limited way in which we've used it is its manageability and reporting capabilities. For example, using simple SQL-NS tools, we could disable certain components of the application while leaving others running. This would allow us to perform incremental hardware and software maintenance without a total shutdown of the system. We could also obtain reports about the exact numbers of events and subscriptions going through the system. Chapter 14, "Administering a SQL-NS Instance," examines how to do these tasks, but even in this simple application, these capabilities are present because of what the SQL-NS platform has provided.

Furthermore, the application can be maintained and extended easily. If we wanted to add a new type of event or a new subscription class, for instance, all we would have to do is add the necessary schema and logic to the ADF and then run SQL-NS tools to update the application (Chapter 5, "Designing and Prototyping an Application," introduces the tools and techniques for doing this). Updating the application would automatically modify existing database entities and create the new ones necessary to support the extensions. If we were building the application from scratch, this type of incremental update would likely be much more difficult.

If you were building the application from scratch, you would also have to do vast amounts of testing to ensure the reliability and security of the execution engine. Using SQL-NS, you can be sure that the execution engine has been designed with reliability and security in mind and has already gone through rigorous testing in these areas.

In summary, although building just the core functionality of the application from scratch might not take that much longer than it does with SQL-NS, all the supporting features that make the application truly rich would be time consuming to build and difficult to get just right. SQL-NS provides these capabilities for free. Using SQL-NS reduces the time to market for your application and increases its reliability, security, and scalability.

EXTENDING THE STOCK APPLICATION

Although the core of the application built here looks much like it would if this were a real application, its inputs and outputs do not. We have been submitting events and subscriptions from hard-coded test files and looking at notification output through text files.

To make this application real, we would have to add a real event provider that read from a real event source. We would build a full subscription management interface to allow users to manage their subscriptions, and we would add support for delivery to real devices. Later chapters show how to do all these tasks.

Apart from input and output pieces, there is work we would do on the core of the application to make it more real. One limitation of the way the application works today is that it stores no state. Therefore, subscribers get notified of every stock price change that is above the price target value they specified in their subscription. If you had a subscription for XYZ stock with a price target of $50, you'd get notified when the stock hit $51, then again if it fluctuated to $51.25, again if it went back down to $51, and so on. For any user, this flood of notifications would quickly become annoying.

To deal with this, the application could store state information about the last price for which each subscriber was notified for a given stock symbol. It could then use this data to filter out price change events that are just intermediate fluctuations. Maintaining state is a core part of the SQL-NS application model, and Chapter 6 covers this in more detail.

Also, all the subscriptions in this chapter's application are event triggered. That is, they fire in response to an incoming event. We could also support scheduled subscriptions: subscriptions that fire at a particular time of day. This would allow you, for example, to enter a subscription that said, "Notify me of the price of stock XYZ at 5:00 p.m. every weekday."

This type of subscription does not fire in response to any given stock price change event. Rather, as stock price events come in continuously during the day, the application would have to keep track of the latest price for XYZ. At 5:00 p.m., it would send you a notification message containing the latest price. This type of subscription is also supported directly in the SQL-NS application model, and in Chapter 6, you'll see how to implement it.

Cleaning Up the Instance and Application

When you're finished examining the instance and application created in this chapter, you can use the following steps to remove them from your system. These steps clean up the databases, Registry keys, and other settings associated with the instance and application.

NOTE

In all the following instructions, when you are instructed to right-click the instance in the Object Explorer, make sure you right-click the `StockBroker` instance in the Notification Services folder, not the `StockBroker` database in the Databases folder.

1. Stop the SQL-NS service by right-clicking the instance in the Notification Services folder in Management Studio's Object Explorer, selecting Stop from the context menu, and then clicking Yes in the confirmation dialog box. Upon success, dismiss the progress dialog by clicking the Close button.

2. Right-click the `StockBroker` instance and select Tasks, and then Unregister. Confirm the unregister operation by clicking Yes in the confirmation dialog box. Upon success, dismiss the progress dialog by clicking the Close button.
3. Right-click the `StockBroker` instance and select Tasks, and then Delete. Confirm the delete operation by clicking Yes in the confirmation dialog box.

Summary

In the SQL-NS application model, both events and subscriptions are just data. SQL-NS uses SQL joins to match the events and subscriptions, and because SQL Server can usually execute these efficiently, notification applications built to this model tend to scale well.

Building an application on SQL-NS involves writing an ADF: an XML document that describes the schemas, logic, and component configuration of the application. This chapter described the ADF for a sample application: one that sends stock price updates to subscribers. The information in this chapter should give you a feel for SQL-NS applications and what's involved in building them. Later chapters provide the details and teach you how to build your own applications.