SDK Programmer's Guide

Last Revised: February 6, 2018

Copyright Statement

Encompass SDK Programmer's Guide

Rev. 2/6/2018

Table of Contents

Section 1

Technical Overview

Why an SDK?

The Encompass Software Development Kit (SDK) provides your organization the ability to integrate your existing back office applications with your Encompass system.

By using the Application Programming Interface (API) included with the SDK, software developers can write custom code to move data between applications, create processes for updating or reporting on your loans, or write their own, stand-alone user interfaces.

What's Included?

The SDK installation package includes the following components.

EncompassObjects API

The EncompassObjects API consists of .NET assemblies (DLLs) that implement the business objects and their methods. The entire public interface of the API is in a single assembly called EncompassObjects.dll. When you write applications for use with the EncompassObjects API, it is this assembly against which you build.

EncompassObjects API Reference

This reference, in the form of a help file named EncompassObjects.chm, provides a complete listing of the objects provided by the API as well as their properties and methods. It also includes coding examples to demonstrate how many of the objects are used. You can access the reference from the Encompass SDK program group in the Start menu after you install the SDK.

Sample Code

Located in the SDK/Samples folder under the installation root, the sample applications demonstrate how to build an application using the EncompassObjects and EncompassAutomation APIs. The sample code is written in C# and VB.NET, and demonstrate use of the API in a standalone app, a web-based app, an Encompass plugin, and a custom input form. You will need Visual Studio 2010 or higher to load, compile and run these projects.

Encompass SDK Programmer's Guide

This document provides an introduction to installing, using, and distributing the SDK.

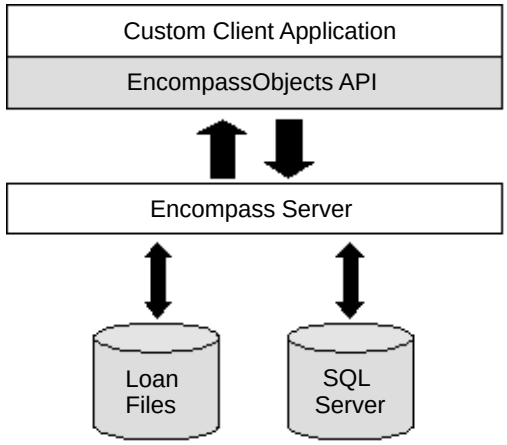
Run-Time Components

The Encompass API is made up of two distinct sets of runtime objects.

- The *EncompassObjects* API, which provides for external applications to interact with an Encompass system either locally or using a remote Encompass Server.
- The *EncompassAutomation* API, which allows for integration directly in the Encompass client application so you extend the functionality of Encompass. (Banker Edition only)

The EncompassObjects API is a data-layer/client-connectivity API, meaning that it provides the functionality you would need to connect your own custom application to an Encompass Server using the same protocols used by the Encompass client. This API also defines the core data structures that form the basis of the Encompass system, such as the Loan and Contact objects. The following diagram depicts a typical run-time environment that uses the EncompassObjects API.

The following diagram depicts a typical run-time environment that uses the EncompassObjects API.



In this model, the custom application manipulates the basic data types that comprise the Encompass system by interfacing directly with the API. The API in turn communicates with the Encompass Server.

The EncompassAutomation API builds on the EncompassObjects API by allowing a developer to manipulate and extend the Encompass Banker Edition user interface. Unlike the EncompassObjects API, code that is written against the EncompassAutomation API runs directly within the Encompass application, either as an *Encompass Custom Form Codebase Assembly* or an *Encompass*

Plugin Assembly. Both DLL types are loaded by the client at runtime and can respond to different events, both internal and external, to affect the desired behavior.

Both APIs are implemented using version 4.5 of Microsoft's .NET Framework. Although, the EncompassObjects API assembly can be used by projects implemented using later versions of .NET, compatibility with all functions cannot be guaranteed. Moreover, because assemblies that reference the EncompassAutomation API are loaded into and executed directly within the Encompass client application, they must follow much more rigid design and implementation guidelines. For more information on how to get started creating projects using the EncompassObjects and EncompassAutomation APIs, see Section 3, "Getting Started" on page 6.

Supported Operations

The EncompassObjects API provides support for a wide variety of tasks. These operations include:

- Opening and modifying existing loans
- Creating new, empty loans
- Assigning loans to a loan officer and/or loan processor
- Importing data from and exporting data to external systems
- Importing data from and exporting data to Fannie Mae-formatted files using built-in methods
- Performing complex queries against the set of all loans
- Enumerating the list of defined loan folders
- Creating new loan folders
- Opening and modifying existing borrower and business contacts
- Creating new contacts
- Performing queries against the contacts database that include criteria related to previously completed loans

The API provides access to the full set of data included in a loan file and automatically performs all the calculations necessary for derived fields. This access includes the ability to add and remove borrower pairs and to add and remove verification records such as employment and residence verifications.

The EncompassAutomation API provides additional support for integration-related tasks, such as:

- Building custom input forms that access remote databases, invoke web services or provide full-featured GUIs that allow for manipulation of loan-related data.
- Monitoring UI-related events from the application such as when the user opens or closes a loan.
- Changing the currently displayed screen based on a user action or external event.
- Launching an Ellie Mae Network™ transaction.
- Accessing and modifying the currently open loan within the UI.

EncompassObjects API

The EncompassObjects API incorporates all of the classes that represent the basic data model of the Encompass system as well as the interfaces to retrieve, update and save those objects to an Encompass Server. This API contains everything you need to write your own custom application that incorporates data from your Encompass system, whether you are writing a GUI-based desktop application, a back office integration application or even a web application that displays Encompass data.

Fundamental to using the EncompassObjects is an understanding of the key objects provided by the API, which are detailed below.

- The ***Session*** object represents a login session with an Encompass Server. Before taking any actions to retrieve or modify Encompass-related data, your code must create and start a Session. Once established, all business objects accessible on the specified server are accessed via the Session.
- The ***Loan*** object represents a single loan. Through the Loan object you can access all of the loan's fields, export and import to supported data file formats, and assign the loan to a specified loan officer or processor.
- The ***Contact*** object provides an abstract base class for both borrower and business-related contacts (represented by the BorrowerContact and BizContact classes, respectively). The Contact object provides access to all of the editable fields associated with the individual, such as name, address, and history information. The Contact object also allows access to the set of notes defined for the individual.
- The objects from the ***EllieMae.Encompass.Query*** namespace provide the object-level interface for performing ad hoc queries against both the loan and contact repositories.

These classes, as well as dozens of others, are located within the EncompassObjects.dll file which ships with both Encompass and the Encompass SDK. Developers should reference this assembly when creating projects that need access to Encompass data. For more information about setting up your development environment to use the EncompassObjects API, see Section 3, "Getting Started" on page 6.

Encompass Automation API (Banker Edition only)

The EncompassAutomation API provides an additional set of classes which represent the visual structure of the user interface. Using these objects you can create plugin-type functionality which extends or modifies the behavior of the client application. Unlike the EncompassObjects API, the EncompassAutomation API cannot be used to author external applications which run outside of Encompass. Instead, the automation interfaces provided by this API are generally used in one of two ways:

- In conjunction with a Custom Input Form, either within the event handler code written with the Input Form Builder or as part of a Custom Input Form Codebase Assembly
- As an Encompass Plugin, which is loaded by the application upon login and can intercept certain UI-level events to perform custom actions within the application.

For example, you can create a custom form which invokes a remote web service and populates the data returned from that service into the loan that is currently opened within the user interface. Or, you could synchronize data with an external database whenever the user saves a loan.

The EncompassAutomation API provides the following primary classes to allow this functionality.

- The ***EncompassApplication*** class, which represents the client application. This class's static methods provide the ability to get or set the currently visible screen within Encompass or to respond to events such as the opening or closing of a loan.
- The ***Form*** class, which represents that base class for all input forms. You can author classes that derive from this base class in order to provide enhanced capabilities to your custom input forms.
- The ***PluginAttribute*** class, which allows you to label a class within your assembly as a plugin which should be instantiated upon login to the Encompass system.

All of these classes can be found in the EncompassAutomation.dll file within the Encompass application folder. Because the EncompassAutomation API must be used in conjunction with the client application, the SDK distribution package does not include this assembly or its dependencies. Developers who wish to create assemblies that use the EncompassAutomation API should install Encompass on their system in addition to the SDK distribution.

Sample Code

The SDK includes sample code for both the EncompassObjects API and the EncompassAutomation API. You can use these samples as a template for designing and building your own custom application, plugin or custom form assembly. To access the sample projects, select the **SDK/Sample Projects** from the Start menu.

To demonstrate the use of the EncompassObjects API in a custom application, the SDK includes a LoanViewer sample project written in C#, VB.NET, and ASP.NET. The application includes a simple login screen and then displays a list of loan folders from which the user can make a selection. Drilling down farther, the user can select a specific loan from the folder and then view and edit a small set of fields from the loan. Although fairly simplistic, the LoanViewer sample provides a good introduction for how to load and manipulate the various objects provided by the API. A second, similar sample project, written as an ASP.NET Web Forms application, provides similar functionality thru a web interface.

The SDK also provides three samples of the EncompassAutomation API, all written in C#. Two samples, the DefaultScreenPlugin and the LoanMonitorPlugin (also available written in VB), demonstrate the plugin capabilities of the automation interfaces. The third sample, SampleInputScreen, demonstrates how to create a custom input form and associate a codebase assembly with it, allowing you to employ the full power of .NET from within your custom forms. Each of these projects includes a Readme.txt file which details the steps you will need to take to properly setup and deploy the plugin or custom form.

Licensing

Your Encompass software license entitles you to unlimited use of the EncompassObjects API within your organization. The EncompassAutomation API is licensed for Banker Edition customers only. Your license does not allow for redistribution of the APIs or any software that uses the APIs.

Because the EncompassObjects API is a licensed product, using this API in an external, custom application requires that any computer on which the API is used have a runtime license registered on it using the "API Runtime CD Key" issued to your company. Runtime licensing is not a requirement for use of the EncompassAutomation API. For more information about EncompassObjects API Runtime licensing, see Section 15, "Distributing Applications Built Using the API".

Section 2

Installation

The SDK is packaged as a separate installation from the client and server applications. It can be installed independently of the other Encompass applications, meaning that it is not required that a computer have the full client installed to use the SDK/API.

The SDK includes all of the components required to develop or run against the EncompassObjects API. For example, if you are using the EncompassObjects within a web application it would be sufficient to install only the SDK on your web server instead of the Encompass Client.

The SDK can be installed on any system running a Windows version supported by the Encompass client application.

SDK License Keys

The Encompass SDK/API is not a freely redistributable product. Every machine on which the SDK/API is to be used, either for development purposes or when deployed, must have a valid license key provided by Ellie Mae. The SDK installation process prompts for this key and verifies it directly with Ellie Mae's servers via the Internet at the time of installation. Until the license is validated, you cannot use the EncompassObjects API on that computer. Copying the API's core files from one machine to another is not permitted.

NOTE: *For more information about license keys and distributing the SDK, see Section 15, "Distributing Applications Built Using the API" on page 58.*

When the API is registered with Ellie Mae, a license file is generated for the current machine and includes machine-specific information including the MAC address(es) of the computer. If you replace the NIC card or otherwise modify the MAC address of the computer, the API may stop working. If this occurs, you must reregister the API with Ellie Mae using the SDK Licensing Tool included in the installation.

Installation Procedure

A simple wizard guides you through the installation process as described below.

To Install the Encompass SDK:

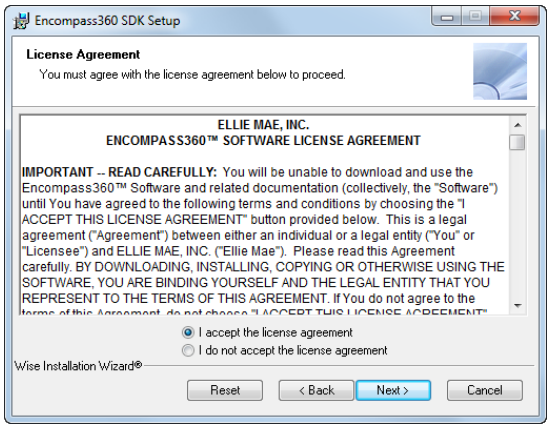
1. Connect to the Internet.

   Connection to the Internet is required to register and activate the SDK.

2. To start downloading the SDK from the Internet, click the link provided to you by your Ellie Mae account representative.
3. Close all Windows applications and click **Next** to begin the installation.

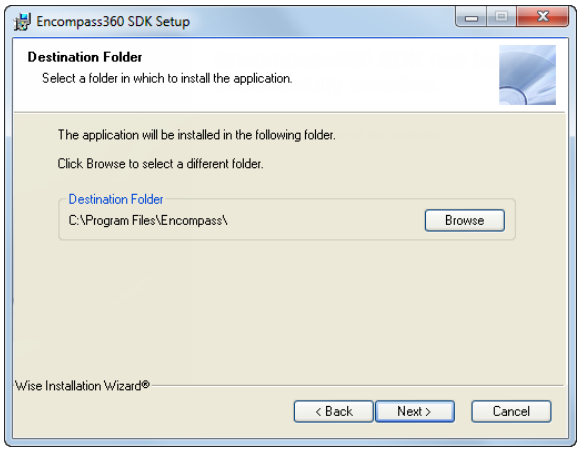
4. On the License Agreement window, read the agreement. If you agree with the terms, select **I accept the license agreement**, and then click **Next**.

   You must accept the license agreement to proceed.



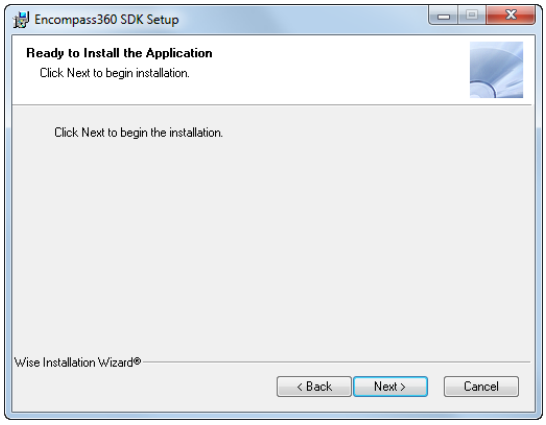
5. On the Destination Folder window, accept the default location and then click **Next**.

To select a different destination, click **Browse**.

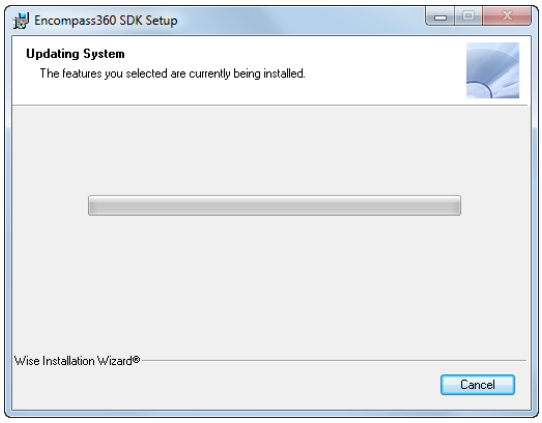


NOTE: *If Encompass is already installed on this machine, the SDK will install in the same location and you will not see this window.*

6. On the Ready to Install the Application window, click **Next** to begin the installation.

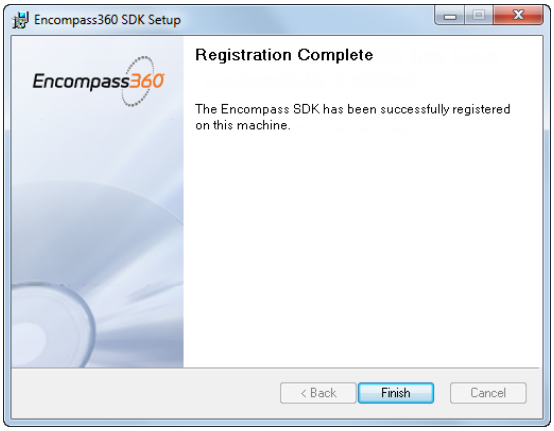


The SDK files will upload to your computer.

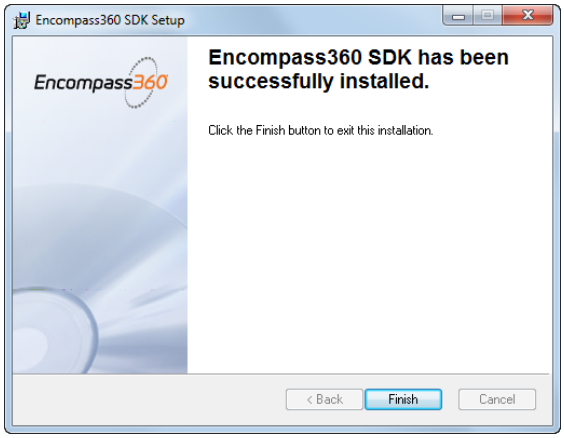


7. Enter the license key you received from your account representative, and then click **Next**.

8. On the Registration Complete window, click **Finish**.



9. On the Installation Complete window, click **Finish**.



Section 3

Getting Started

The steps for creating and deploying a project that uses either of the APIs are dependent both on the type of project you are creating as well as the development environment in which you will do your coding. Additionally, because the Encompass application itself supports multiple deployment methods, ensuring that your application will run correctly in every environment requires that you take special steps that are normally not required when using a third-party component within your application.

The sections below describe the process of creating and deploying your application or component. Each section is written to provide step-by-step instructions tailored specifically to your development environment and project type.

If you are writing an external application, such as a custom desktop application, web application or back-office utility that requires access to the Encompass data store, you will be using the EncompassObjects API. The section *Writing Standalone Applications using EncompassObjects* provides instructions on how to quickly get your project up and running. Additionally, be sure to read the section on Deploying an EncompassObjects Application to ensure that your application will run correctly regardless of how the user runs Encompass and to ensure your application is upgrade-ready when new releases are made.

If you are writing a Plugin Assembly or a Custom Input Form Codebase Assembly, then you will be using the EncompassAutomation API (in addition to the EncompassObjects). These components run directly within Encompass and must be implemented following a stricter set of guidelines than when writing a standalone application. For step-by-step instructions on how to start a plugin or codebase assembly project, skip to the corresponding section below.

Writing Standalone Application Using EncompassObjects

This section describes the process of creating a standalone application using the EncompassObjects API. A standalone application is any process that will run separate from the Encompass client application but which needs to access the data repository. Because the EncompassObjects API requires .NET 4.0, you will need to create your application using any edition of Visual Studio 2012 or later. You may also choose any .NET-compatible programming language such as C#, VB.NET, J#, C++, etc. However, the instructions and examples in this document are limited to the two most common languages, C# and VB.NET.

Step 1: Create your project

To begin your application, launch Visual Studio and start a new project of the appropriate type (Windows Forms, WPF, Console, ASP.NET, etc.). Be sure to set your project's properties to use ".NET Framework 4.5" as the Target Framework. It is not sufficient to use the "Client Profile", which is the default target framework for certain project types. Note also that your choice of project type will impact how you carry out some of the steps below.

Step 2: Add references to the assemblies
To use the EncompassObjects API, you must add a reference to three separate assemblies that are shipped with the SDK: **EllieMae.Encompass.AsmResolver.dll, EllieMae.Encompass.Runtime.dll** and **EncompassObjects.dll**. All three are found in the folder in which you installed the SDK.

The first two of these assemblies provide the interface and implementation of the *API Runtime Services*, a feature that allows programs that use the EncompassObjects API to dynamically locate and load the core API DLLs at runtime. If you follow the instructions in the section *Deploying an EncompassObjects Application*, the Runtime Services will help ensure that your application continues to work properly after an Encompass software upgrade.

After adding the three assemblies as references to your project, be sure to set the *Copy Local* property of the EncompassObjects assembly to **False**. This setting will prevent the assembly (and its many dependencies) from being copied to your application's build folder. Only the EllieMae.Encompass.AsmResolver and EllieMae.Encompass.Runtime assemblies should load from your application's bin folder.

Step 3: Enable the Legacy Code Access Security Policy
The EncompassObjects API and API Runtime Services use the Code Access Security (CAS) policy model that was deprecated as part of the .NET 4 release. To use the EncompassObjects, you must enable support for this legacy CAS model by updating either the app.config or web.config file for your application.

If your application includes an app.config file (e.g. Windows Forms, WPF, etc.), you can enable the legacy CAS policy by adding the NetFx40_LegacySecurityPolicy element to your config file:

```
<configuration>
  <runtime>
    <NetFx40_LegacySecurityPolicy enabled="true"/>
  </runtime>
</configuration>
```

If your application includes a web.config file (e.g. ASP.NET), you enable the legacy CAS policy by adding or modifying the <trust> element in your config file:

```
<configuration>
  <system.web>
    <trust level="Full" legacyCasModel="true"/>
  </system.web>
</configuration>
```

Failure to enable the Legacy CAS policy will cause an Exception when your application first attempts to use the EncompassObjects assembly.

Step 4: Add code to initialize the Encompass API Runtime Services

Within your application, prior to invoking any method from the EncompassObjects API, you must initialize the EncompassObjects Runtime Services. This is done using the following method call:

```
[C#]
new EllieMae.Encompass.Runtime.RuntimeServices().
  Initialize();

[VB.NET]
Dim EncRuntimeSvcs as new
EllieMae.Encompass.Runtime.RuntimeServices()
EncRuntimeSvcs.Initialize()
```

Because you will not be deploying the EncompassObjects assembly with your project (see Deploying an EncompassObjects Application below), it is critical that this method be invoked before the .NET JIT compiler attempts to bind to the EncompassObjects assembly at runtime. Otherwise, your application will fail as the .NET runtime will not be able to locate the EncompassObjects assembly.

The safest way to ensure that .NET will initialize the Runtime Services prior to loading the EncompassObjects is to put the call to the RuntimeServices.Initialize() method as early in the application lifecycle as possible. For example, a Windows Forms application could put this call in the Main() function prior to invoking Application.Run(). Similarly, a WPF application could place the call in the Application_Startup event, and an ASP.NET application could use the Application_Start() event handler in global.asax.

If you find that you must invoke the EncompassObjects assembly in the same function that also initializes the API Runtime Services, you will need to move your code that references the EncompassObjects into a separate function or subroutine. Otherwise, the JIT compiler will attempt to load and resolve the EncompassObjects assembly prior to invoking RuntimeServices.Initialize().

The following example shows how a Windows Forms application could be written to initialize the Runtime Services in the application's Main() routine while all previous logic from Main() was moved to a separate function.

```
[C#]
public static void Main()
{
  new EllieMae.Encompass.Runtime.RuntimeSer-
    vices().Initialize();
  startApplication();
}

private static void startApplication()
{
  // Place the prior contents of your Main() function here.
}

[VB.NET]
Public Sub Main()
  Dim EncRuntimeSvcs as new EllieMae.Encompass.Run-
    time.RuntimeServices()
  EncRuntimeSvcs.Initialize()
  StartApplication()
End Sub

Private Sub StartApplication()
  'Place the prior contents of your Sub Main here
End Sub
```

Depending on how you use the EncompassObjects within your application, you may not find that it is necessary to use the technique described above, in particular if you invoke the EncompassObjects only within an isolated region of your application which is not referenced at startup. However, you should ensure that, when deployed without the EncompassObjects.dll file, your application continues to function correctly.

NOTE: *if you find that Visual Studio gives you a compile error indicating that the EllieMae namespace could not be found, check your project's properties to ensure that you have selected ".NET Framework 4.5" as the Target Framework. This error can occur when the "Client Profile" version of the .NET Framework 4.5 is selected, and the EncompassObjects API is not compatible with that version.*

Step 5: Connect to the Encompass Server

You are now ready to create your initial connection to the Encompass Server to begin sending and receiving Encompass data. Because the EncompassObjects API is secured using an Encompass User ID and password in the same manner as the client application, you will need to either hard-code a User ID/Password pair into your application (appropriate only for applications being built for in-house use) or prompt the user for their ID and password. You will also need to determine, either by hard-coding or prompting the user, the name/URL of their Encompass Server.

Once you have this information, you can create your Encompass Session as follows:

```
[C#]
using EllieMae.Encompass.Client;

private Session ConnectToServer()
{
  Session s = new Session();
    s.Start("https://be11111111.ea.elliemae.net$be11111111",
    "mary", "maryspassword");
}

[VB.NET]
Imports EllieMae.Encompass.Client

Private Function ConnectToServer() as Session
  Dim S as New Session
  S.Start("https://be11111111.ea.elliemae.net$be11111111",
    "mary", "maryspassword")
  Return S
End Function
```

See the *"Connecting to an Encompass Server"* section on *page 9* for additional information.

Step 6: Compile and test your application
Assuming that you have set the **CopyLocal** property of the EncompassObjects assembly to **false** as instructed above, your project's build folder should contain only your assemblies and the two additional, version-independent Encompass assemblies: **EllieMae.Encompass.Runtime.dll** and **EllieMae.Encompass.AsmResolver.dll**. Any additional Encompass assemblies should be removed as they will prevent the API Runtime Services from ensuring that a consistent set of assemblies is loaded into your project.

If you encounter an error when attempting to run your application and you believe the source to be the EncompassObjects API or the API Runtime Services, refer to Appendix B, Troubleshooting the EncompassObjects API, for a list of common errors, their causes and resolutions.

Step 7: Deploy your application
When it comes time to deploy your application or to package it for distribution or sale, special considerations need to be taken to ensure that your application will function correctly regardless of the way Encompass is installed and regardless of the end user's version. Read the "Deploying an EncompassObjects Application" section on page 8 for more information on how to ensure your application is deployed correctly.

Deploying an EncompassObjects Application

Whether you are deploying your application to your users' desktops, to a web server or packaging it for retail, understanding how to properly deploy your EncompassObjects-based application is critical. Failure to follow the guidelines below can lead to your application becoming inoperable within certain customer environments or when you (or your customer) upgrade to a new version of Encompass.

When you package your application, you must follow these rules:

Rule #1: Include only the API Runtime Services DLLs
The SDK ships with a large number of DLLs that are required for the execution of your application. However, you should package or deploy only two of these DLLs with your application:

EllieMae.Encompass.Runtime.dll

EllieMae.Encompass.AsmResolver.dll.

These two DLLs are version independent and will work with future versions of Encompass. They contain the necessary logic to ensure that your application can find and use the other, version-specific DLLs already installed on the user's computer. In particular, **do not** deploy the EncompassObjects.dll or any of its dependencies with your application.

Rule #2: Place Runtime Service DLLs in the same folder as your EXE
The two DLLs mentioned in Step #1 should be installed alongside your application, in the same folder as your program's executable (EXE). If you are deploying an ASP.NET application, these assemblies should be placed in your application's bin folder with your other compiled assemblies.

Rule #3: Ensure the target machine has the Encompass SmartClient or SDK installed
Because you are not shipping the EncompassObjects assembly or its dependencies with your application, each machine on which it runs must have either the Encompass SmartClient or Encompass SDK installed.

In general, you should avoid installing the SDK on machines that already have the Encompass SmartClient unless the machine is used for development purposes. The API Runtime Services will prioritize the assemblies installed with the SDK ahead of those installed by the SmartClient when loading assemblies at runtime. This behavior can cause problems when the Encompass Server is upgraded to a new version of Encompass. When a server upgrade occurs, the Encompass SmartClient automatically downloads and installs the updated API components for that version. As a result, if your application utilizes those assemblies thru the Runtime Services, it will also be upgraded without the need for you to intervene.

On the other hand, if you install the Encompass SDK on the machine, you will be required to upgrade the SDK to the correct version of Encompass with each Encompass server update. Because the EncompassObjects assembly is version-dependent, it will only work with the same version of the Encompass Server service. Until the SDK is upgraded (or uninstalled), your application will not be permitted to connect to the upgraded Encompass Server.

Connecting to an Encompass Server

The first step in almost all projects that interact with the API's object model is to establish a connection to an Ellie Mae-hosted Encompass Server. The core class for establishing either kind of connection is the Session object (found in the EllieMae.Encompass.Client namespace). The Session object represents a single, logical connection to an Encompass system and it is through the Session instance that you access all of the Encompass data required by your application.

To begin a session, you must generally have three pieces of information: a user ID, a password, and the URI (Uniform Resource Identifier) of the server to which you wish to connect. The server URI provides the Session with the name of the Encompass Server and the other details required to make an initial connection. The URI has the following format:

"https://BE11111.ea.elimae.net$BE11111"

When you log into an Encompass Server, you are required to provide a valid Encompass user ID and password. The user under which the session is established influences the way in which the API behaves, both in terms of the data provided to the caller as well as the properties and methods the application invokes. For example, the Folder object (in the EllieMae.Encompass.BusinessObjects namespace) provides a method named GetContents() that returns the iden-tities of all loans in the folder that are visible to the logged in user. Thus, different logins can result in different lists of loans from this call.

Similarly, if an attempt is made to retrieve a loan using its GUID (via the OpenLoan method on the Loans class), an exception is raised if the logged in user does not have the necessary rights to access the loan.

Once you have the necessary information to connect to the server, your code should instantiate a new Session object and invoke either its Start() method or its Start Offline() method.

```
Session s = new Session();
s.Start("https://be11111111.ea.elliemae.net$be11111111",
"mary", "maryspassword")
```

Any number of sessions can be concurrently opened to any number of Encompass Servers, each with its own URI and login information. Once a session is established, all subsequent access to the server comes directly or indirectly through the Session object. For example, to retrieve a loan from the server, first access the Loans property from the Session object.

```
Loan loan = Session.Loans.OpenLoan(…)
```

Because the Encompass Server is built to support connected sessions (sessions that connect once and perform multiple data transactions over their lifetime), you should generally open your connection early in your application and close it only when you have completed your interactions with the server (or if there will be a long period of inactivity). Storing the Session object in a global variable to maintain an easily accessible reference is often a good idea.

When you complete your interaction with the Encompass Server, use the Session's End() method to log out of the server and release your session's resources. If you do not invoke the End() method, you will unnecessarily waste server resources until the server cleans up the session. This occurs when your application terminates or when the Session object is garbage collected by the .NET framework.

Creating an Encompass Plugin Assembly (Banker Edition only)

An Encompass Plugin is a .NET assembly which is loaded by the client application when a user logs in, and it remains resident and active for the entire duration of the client session. The primary function of a plugin is generally to sit dormant and wait for specific events, either from inside the Encompass application or from an external source, that trigger the plugin's functionality.

Some examples of functions that can be built using Encompass plugins:

- Monitoring when a user changes a specific field in a loan and, when it occurs, perform custom calculations or business logic.
- Synchronizing data from a loan to an external database when the loan being edited by the user is saved.
- Responding to events from an external application, such as a call center application, by using the built-in DataExchange capabilities of the API.

An Encompass Plugin Assembly is any .NET assembly that has one or more classes which carry the EllieMae.Encompass.ComponentModel.PluginAttribute attribute. To create such an assembly in Visual Studio 2010, follow these steps:

1. Create a new Class Library project using any .NET-compatible programming language of your choice. For the samples below, the language is assumed to be C#, but you may choose VB.NET, J#, C++.NET, and so on. Set the project's properties to use a full version of the .NET Framework (v4.0 or lower). Do **not** use the "Client Profile" version of .NET.

2. In the project's References, add the files EncompassObjects.dll, which is found in the folder where the full SDK is installed, and the EncompassAutomation.dll, which can be found in the Encompass Client installation path under the "SDK" folder on any machine where the Encompass Client is installed. The full SDK does not in stall the EncompassAutomation.dll. The "CopyLocal" attribute should be set to True for both references.
3. Create your primary plugin class and assign it the Plugin attribute.

```
using EllieMae.Encompass.ComponentModel;
[Plugin]
public class MyPlugin
{
}
```

4. Create a public, parameterless constructor for your plugin class. This is a requirement for every plugin class. Within your constructor, you may subscribe to one or more of the events provided by the EllieMae.Encompass.Automation.EncompassApplication class or the EllieMae.Encompass.Client.Session class. The following code subscribes to the event that is fired when a loan file is opened in the Encompass user interface.

```
public MyPlugin()
{
  EncompassApplication.LoanOpened +=
    new EventHandler(Application_LoanOpened);
}
```

5. Create your event handlers, which will perform the desired action(s) when the event is triggered. For example, the following code prompts the user if they would like to jump to the Transmittal Summary form if they open a loan which is already in the Completed state.

```
private void Application_LoanOpened(object sender,
  EventArgs e)
{
  MilestoneEvent ms = EncompassApplication.
    CurrentLoan.Log.MilestoneEvents.GetEvent-
    ForMilestone("Completion");
  if (ms.Completed)
  {
      DialogResult res = MessageBox.Show
        (EncompassApplication.Screens,
        "Go to Transmittal Summary?", "Plugin",
        MessageBoxButtons.YesNo);
      if (res == DialogResult.Yes)
        Macro.GoToForm("Transmittal Summary");
  }
}
```

6. Build your assembly.
7. Upload the resulting DLL to the Encompass Server by using the Input Form Builder.
   - Within the Input Form Builder, navigate to the Tools > Manage Customizations > Plugins tab, and then click the **Upload** button to upload the plugin DLL to the Encompass server. For more detailed instructions, view the *Encompass Input Form Builder User's Guide* and refer to *Chapter 11, Managing Your Forms, Plugins, and Custom Data Objects*.
8. Alternatively, as an advanced solution, the SDK itself can be utilized to automate the upload of the plugin. An Ellie Mae Form Package (.EMPKG) and corresponding manifest.xml must be created, and this package can then be uploaded through the XferPackageImporter class of the SDK. For detailed instructions, visit the Ellie Mae Knowledge Base article #9716 *"Using the SDK to automate the upload of plugins"*.
9. Launch Encompass and log in. Your plugin will be automatically loaded.

To remove a plugin from an Encompass system, use the Input Form Builder the same way as uploading the plugin, but click the **Delete** button in the Plugins tab. For more detailed instructions, view the *Encompass Input Form Builder User's Guide* and refer to *Chapter 11, Managing Your Forms, Plugins, and Custom Data Objects*.

The SDK ships with three sample plugin projects, all written in C#. These projects can be found in the SDK\Samples\C# subfolder of your SDK installation directory (typically C:\Program Files\Encompass). The project names are DefaultScreenPlugin, LoanMonitorPlugin, and DataExchangeExample.

Creating a Custom Form Codebase Assembly (Banker Edition only)

Custom Form Codebase Assemblies are used to extend the functionality of your custom input forms created using the Input Form Builder. By associating a codebase assembly with a custom input form you can provide far more sophisticated interfaces and behaviors than are possible by simply using the native abilities of the Input Form Builder.

As with plugin assemblies, custom codebase assemblies must be written against version 4.5 of the .NET Framework.

Section 12, "Custom Input Form Codebase Assemblies (Banker Edition only)" on page 51 provides extensive details of how the codebase assemblies work, provides example code and discusses typical use cases. What follows are the steps you will need to take within the VS.NET 2012 development environment to create a codebase assembly.

1. Create a new Class Library project using any .NET-compatible programming language of your choice. For the samples below, the language is assumed to be C#, but you may choose VB.NET, J#, C++.NET, and so on.

Set the project's properties to use a full version of the .NET Framework (v4.5). Do **not** use the "Client Profile" version of .NET.

2. In the project's References, add the files EncompassObjects.dll, which is found in the folder that the full SDK is installed, and the EncompassAutomation.dll, which can be found in the Encompass client install path under the SDK folder on any machine that has the Encompass client installed. The full SDK does not install the EncompassAutomation.dll.
3. Create your form's codebase by creating a new class that derives from EllieMae.Encompass.Forms.Form.

```
using EllieMae.Encompass.Forms;
public class DemoForm : Form
{
}
```

Note that the name of the class is not important and does not need to match the name of the input form for which it will act as the codebase. In fact, a single class can potentially act as the codebase for multiple forms.

4. Declare instance variables for the controls for which you will be catching events. For example, if your custom input form contains a button named "btnQuote" for which you will handle the Click event, declare a Button control of the same name.

```
using EllieMae.Encompass.Forms;
public class DemoForm : Form
{
  private Button btnQuote = null;
}
```

5. Next, attach your custom event handlers to the controls' events by overriding the CreateControls() method. The FindControl() method is used to retrieve the underlying control object from the underlying Form.

```
using EllieMae.Encompass.Forms;
public class DemoForm : Form
{
  private Button btnQuote = null;
  public override void CreateControls()
  {
    this.btnQuote = (Button)
      FindControl("btnQuote");
    this.btnQuote.Click +=
      new EventHandler(btnQuote_Click);
  }
  private void btnQuote_Click(object sender,
    EventArgs e)
  {
    using (QuoteDialog frm = new QuoteDialog
      (this.Loan)) frm.ShowDialog
      (EncompassApplication.Screens);
  }
}
```

6. Compile your codebase assembly into a DLL.
   - Encompass runs as a 32-bit application on a 64-bit Operating System. If compiling the DLL in Visual Studio, the 'Target platform' should be set to *ANY CPU*.
7. Launch the Input Form Builder and open your existing custom form (or create a new one). Your form should include controls with Control IDs matching the names used when calling FindControl() method. In the example above that would mean your form should have a Button control with the Control ID set to "btnQuote". This control can appear anywhere on the input form.
8. Select the Form object from the Properties dropdown list so its properties are visible. Choose the CodeBase property and click the "…" to launch the Codebase Selector. Browse to the location of your codebase assembly and select it. Encompass will automatically search the assembly for any classes which derive from the EllieMae.Encompass.Forms.Form class and list them in the dropdown. Select your form's codebase class and click OK.
9. Save your input form, assigning a new name if necessary.
10. Launch Encompass and open your custom form. When you trigger an event which you have elected to capture within your custom codebase assembly, your code will be automatically invoked.

The code above is taken from a sample custom codebase assembly which is included with the SDK. This project is located in the SDK\Samples\C#\SampleInputScreen folder beneath the folder you selected when the SDK was installed (typically C:\Program Files\Encompass). This project includes a sample custom input form named *InsuranceForm.emfrm* which can be used in conjunction with the sample codebase assembly. Read the *readme.txt* file which accompanies this project for details on compiling and demonstrating this assembly.

Section 4

Managing Loans

The EncompassObjects API provides a number of classes used to locate, manipulate, and update the data within a loan. This section discusses the most important of these classes and provides brief code examples to demonstrate their use.

Retrieving, Saving, and Locking Loans

Just as all communications with the Encompass Server must pass through the Session object, all loan-related activity passes through the Loans object, available as a property off of the Session. The Loans object provides the necessary methods to open existing loans, create new loans, and perform queries to select subsets of loans based on ad hoc criteria. The following code opens an existing loan (if you know the loan's globally unique GUID).

```
Session s = new Session();
s.Start("http://be11111111.ea.elliemae.net$be11111111",
  "mary", "maryspwd");
Loan loan = s.Loans.Open("{AA6C8474-05D8-3D32-BC9A-
  36C1D2C6DB33}");
```

Alternatively, if you do not know the GUID, you can use a LoanIdentityList, and the name of the folder in which it resides. The LoanIdentifyList will provide the GUID for which to open the loan.

```
Session s = new Session();
s.Start("myserver", "mary", "maryspwd");
LoanFolder folder = s.Loans.Folders["My Pipeline"];
LoanIdentityList ids = folder.GetContents();
foreach (LoanIdentity id in ids)
{
  Loan loan = folder.OpenLoan(id.LoanName);
};
```

Note that the loan name specified is not a field that is publicly visible to users of the Encompass application. For Encompass 16.2 and later, the loan name is the same as the loan's GUID. In addition, the property can't be set and it shouldn't be used in any loan open calls.

Once you retrieve a Loan object from the server, you can manipulate it in memory until you are ready to save its contents back to the server. The topic below, entitled Loan Fields, discusses how your application can retrieve and modify the data within a loan. Saving the loan is generally very simple: the Loan object has a Commit() method that posts the new loan data back to the server.

```
Session s = new Session();
s.Start("myserver", "mary", "maryspwd");
Loan loan = s.Loans.Folders["My
  Pipeline"].OpenLoan("Ebberson");
loan.Lock();
… // Make changes to the loan's contents
loan.Commit();
loan.Unlock();
```

Note that the code also invokes the methods Lock() and Unlock() on the loan. Although you can manipulate the data within the loan without first calling the Lock() method, you must hold a lock on the loan in order to commit your changes to the server.

Lock() and Unlock() provide a simple means of ensuring data consistency in an environment in which multiple client applications can access the same loan. Invoking the Lock() method causes the Encompass Server to grant the currently logged in user exclusive write access to the loan. If another client application subsequently attempts to invoke Lock() for the same loan prior to the current process calling Unlock(), an exception is raised. Note that the Lock() method is non-blocking – thus, when calling the Lock() method, you should always be prepared for the possibility that an exception will be raised because of a lock held by another user.

Once a lock is acquired, no other process can modify the loan's contents on the server until you unlock it. Failure to invoke the Unlock() method after successfully calling Lock() causes the lock to be held open until explicitly closed by an administrator from within the Encompass application. Maintaining locks across process or session lifetimes can be useful under certain circumstances and the Encompass Server does nothing to prevent this.

Creating and Assigning Loans

Besides manipulating the data within existing loans, the EncompassObjects allow client applications to create new loans to be managed by Encompass. Creating a new, empty loan is simple.

```
Loan newLoan = session.Loans.CreateNew();
```

Note that by calling the CreateNew() method you simply create a new in-memory representation of a loan. This loan does not yet exist on the Encompass Server and will not exist until you invoke the Loan's Commit() method.

However, before calling Commit you must first indicate the Loan Folder into which the loan will be saved.

```
newLoan.LoanFolder = "My Pipeline";
```

If you do not set this property, or if you set this property to a non-existent folder, the Encompass Server will throw an exception when you call Commit().

Although there is a LoanName property, starting with Encompass 16.2 and later, it is not allowed to be set. By default it is the same as the loan's GUID, ensuring that the LoanName is always unique. You can try to set it using the SDK, but it will not change.

Once a loan is created (or at any point in the loan's lifetime), you can assign the loan to a particular loan officer or send the loan to processing. The Loan object provides two methods to accomplish these tasks: SendToLoanOfficer() and SendToProcessing(). Each method requires that you provide the User (refer to Section 8, "Managing Users" on page 39) to whom the loan should be directed. Just as with all other changes to the loan, the effect of invoking these methods is not made permanent until you invoke the loan's Commit() method.

Loan Fields

The Loan object's properties and methods deal primarily with reading and writing the values of hundreds of loan fields. Every piece of data held within the loan has a unique Field ID you can use to access the field's value and metadata. Some Field IDs are purely numeric, such as the street address of the subject property (field "11"). Other Field IDs can contain non-numeric characters such as "VEND.X133", which represents the name of the listing agent for the property.

Fields also come in four different varieties.

- **Simple fields:** Only a single instance of a simple field appears within a loan. Examples of simple fields are the subject property address fields.
- **Multi-valued fields:** Some fields represent data on forms for which there can be multiple instances, such as the line items on the Trust Account Form. A single, fixed Field ID is not sufficient to identify the data which is being referenced.
- **Borrower Pair-specific fields:** Encompass supports the ability to define multiple borrower pairs within a single loan. Because some forms are specific to a certain borrower or borrower pair, it is necessary to identify for which pair the specified field is being requested.
- **Borrower Pair-specific, multi-valued fields:** These fields are both specific to a single borrower pair and can take on multiple values for a single pair. For example, the verification of employment fields, which are specific to a borrower but for which a single borrower may have multiple records.

For a complete list of the loan fields available in Encompass and their Field IDs, refer to the *Encompass Field ID / Model Path Mapping* knowledge base article (# 000005032) in the Ellie Mae Resource Center.

Field Formats and Descriptors

Every field is assigned an explicit field format that defines the data type of the field (such as integer or string) and the display format for the field (e.g. the number of decimal places to display). The following list contains the field formats used within Encompass:.

Format	Data Type	Display Format/ Restrictions
STRING	String	None
YN	String	"", "Y" or "N"
X	String	"" or "X"
ZIPCODE	String	nnnnn-nnnn
STATE	String	Two-character abbreviation
PHONE	String	nnn-nnn-nnnn nnnn
SSN	String	nnn-nn-nnnn
INTEGER	Int32	n,nnn
DECIMAL	Decimal	n,nnn.n...
DECIMAL_1	Decimal	n,nnn.n
DECIMAL_2	Decimal	n,nnn.nn
DECIMAL_3	Decimal	n,nnn.nnn
DECIMAL_4	Decimal	n,nnn.nnnn
DATE	DateTime	MM/dd/yyyy
MONTHDAY	DateTime	MM/dd
DROPDOWN*	String	None
DROPDOWNLIST*	String	Only values specific in the options list allowed.
AUDIT*	String	Read-only

* These fields are available only for custom field types. Encompass enforces the field format specification when you are setting the value of any field. For example, if you attempted to set the value of a field which has the INTEGER format to a string such as "ABC", Encompass will raise an exception.

To determine the format of a field you can use the *field descriptor* associated with the field. A field descriptor provides metadata for the individual field, including format, description and, if appropriate, the set of allowed values which the field can take on. You can access a field's descriptor through the Loans.FieldDescriptors property.

```
FieldDescriptor des = session.Loans.Field
  Descriptors["1109"];
if (des.Format == FieldFormat.INTEGER)
  loan.Fields["1109"].Value = 10;
```

For fields that are restricted to a particular set of values, use the descriptor's Options collection to determine what the allowed values are. For example, the following code would output all of the values permitted in the Loan Purpose field (field 19).

```
FieldDescriptor des = session.Loans.Field
  Descriptors["19"];
foreach (FieldOption option in des.Options)
  Console.WriteLine(option.Value);
```

Encompass custom fields which have a pre-defined list of allowed values will have a field format of DROPDOWN or DROPDOWNEDIT. Standard fields, however, do not use these field types even if they include a set of pre-defined values. For example, every field with the YN format has two allowed options: "Y" and "N". The empty string is always considered an allowed value for every field since this represents the default, unset field state.

Accessing Simple Fields

The Loan object includes a Fields collection that provides direct access to the loan's data using Field IDs. For example, the following code accesses a simple field, the subject property address.

```
string addr = loan.Fields["11"].FormattedValue;
```

When accessing a field's value you have three choices as to how you would like the value returned to you, depending on whether you use the Value property, FormattedValue property, or UnformattedValue property:

Value Property
The **Value** property returns the field's value in its native type. For example, a field which is an INTEGER field will return an integer while a field which is a DATE field will return a DateTime value. If the field has not yet been set, the Value property will return null (except in the case of string-based fields, which will return an empty string when unset).

```
Decimal loanAmt = (Decimal) loan.Fields
  ["1109"].Value;
```

Note that this example assumes the field's value is set. If the field were unset, this code would cause an exception since the Value property would return null. To prevent this, you can use the field's IsEmpty() method to detect if the field value is set.

```
Decimal loanAmt = 0;
if (!loan.Fields["1109"].IsEmpty())
  loanAmt = (Decimal) loan.Fields["1109"].Value;
```

Alternatively, you can use the field's ToDecimal() function to force the field to be converted to a Decimal value. If the field is unset, the ToDecimal() function returns 0.

```
Decimal loanAmt = loan.Fields["1109"].ToDecimal();
```

FormattedValue Property
The **FormattedValue** property, which returns a formatted string representation of the field's value. For example, a field that has format DECIMAL_2 and is set to 5203.5 would return a FormattedValue of "5,203.50". For phone, zip code or other formatted string types, the FormattedValue property returns the field value with all of the formatting characters in place.

```
string amtString = loan.Fields["1109"].FormattedValue;
```

UnformattedValue property
The **UnformattedValue** property returns a string representation of the field's value with formatting characters removed. For example, a phone number field which contains the value "555-555-3434" would have an UnformattedValue of "5555553434".

If you are using a COM-based application with the EncompassObjects API, be aware that the Value property behaves differently in this environment. Because of issues marshalling the Object type across the COM-.NET boundary, the Value property will always return a string value (in fact, it will always return the FormattedValue). To force the field value to its native data type, use the ToInt(), ToDecimal() or ToDate() methods.

Accessing Multi-Valued Fields

You can access multi-valued fields in two ways. The first is to use the GetFieldAt() method on the LoanFields object as follows.

```
string value = loan.Fields.GetFieldAt("TADS", 2)
```

The above code retrieves the Description field from the second line item on the Trust Account Form. Note that multi-valued field's indices are 1-based, not 0-based.

The second way to access the same field as in the sample above is to generate the instance-specific Field ID manually. Each value in a multi-values field has its own, unique Field ID with the following format.

```
<2-Character Parent Field Prefix><2-Digit Index><Parent
  Field Suffix>
```

You can also directly access the field in the example above as follows.

```
string value = loan.Fields["TA02DS"].FormattedValue;
```

In this example, “TA” represents the prefix for the field, “02” is the 2-digit instance index and “DS” is the field suffix for the Description field. The field prefix is always the first two characters of the multi-valued Field ID, with the suffix composed of the remaining characters.

Accessing Borrower Pair-Specific Fields

Unlike multi-valued fields, borrower pair-specific fields do not have their own unique Field IDs to access the multiple instances of the data. You have two choices in how to access the data for a specific borrower pair. Each loan always has set within it a “current” borrower pair. This is the pair that will be used for form generation when forms are printed through the Encompass application.

To set the field values for the current borrower pair, reference the field directly. For example, to access the first name of the primary borrower (Field ID = 36) for the current borrower pair, use the following code.

```
string value = loan.Fields["36"].FormattedValue;
```

Note that for a typical loan that only contains a single borrower pair, this is all that is required since that pair must also be the current pair.

If you need to access the data for a different borrower pair, you can make that pair the current pair,

```
BorrowerPair newPair = loan.BorrowerPairs[2];
loan.BorrowerPairs.Current = newPair;
string value = loan.Fields["36"].Value;
```

or use the methods provided on the LoanField object to retrieve the value for the desired pair.

```
BorrowerPair newPair = loan.BorrowerPairs[2];
String value = loan.Fields["36"].GetValueFor
  BorrowerPair(newPair);
```

Either technique requires that you first retrieve the borrower pair of interest from the loan’s BorrowerPairs collection. This collection provides access to all of the defined borrower pairs and permits creation of new pairs, deletion of existing pairs, and setting or retrieving of the current pair.

Accessing Borrower Pair-Specific, Multi-Valued Fields

Borrower pair-specific, multi-valued fields, such as those on the verification forms, are accessed using a combination of the multi-valued and borrower pair-specific techniques referenced above. For example, the following code accesses the name of the employer from the third VOE instance for the second borrower pair.

```
BorrowerPair newPair = loan.BorrowerPairs[2];
String value = loan.Fields.GetFieldAt("BE02", 3).
  GetValueForBorrowerPair(newPair);
```

Dealing with multi-valued fields requires not only special methods to access instances of the fields, but also the means to add, remove, and enumerate these instances.

The Loan object provides object-valued properties for each of these verification types to allow the client application to add, remove, and enumerate the corresponding verification records.

For example, the following code creates a new Verification of Employment (VOE) record for the current borrower pair by invoking the Add() method on the loan’s BorrowerEmployers sub-object. In a similar manner, you can access the collection of each different type of verification records (such as Verification of Deposit and Verification of Liability) through a corresponding property from the Loan object. In all cases, the return value of the Add() operation provides the index of the newly-created verification record, which is used in this case to set the name of the employer for the new VOE. Finally, the code demonstrates how to iterate through the employment verification records, printing the Employer Name field (Field ID = BE02) for each.

```
int index = loan.BorrowerEmployers.Add(false);
loan.Fields.GetFieldAt("BE02", index).Value = "Megacorp
  Industries";
for (int i = 1; i <= loan.BorrowerEmployers.Count; i++)
  Console.WriteLine(loan.Fields.GetFieldAt("BE02",i).Value+
    "");
```

Importing and Exporting Loans

The EncompassObjects API provides full access to loan file data. Therefore, developers can build functionality to import data from and export data to external systems that also support data import and export.

NOTE: *The following section discusses importing and exporting Fannie Mae loan files. For a full list of the additional export formats supported by the Encompass API, refer to the* LoanExportFormat *topic in the “EncompassObjects API Reference” document (help file) that accompanies the SDK. You can access this document from the Encompass SDK program group in the Start menu in the SDK.*

Fannie Mae 3.x Formatted Files

The API includes native import and export methods for use with Fannie Mae 3.x-formatted files. To import a formatted loan file into an existing loan, use the Loan object’s Import method.

```
Loan loan = session.Loans.Open("{AA6C8474-05D8-3D32-
  BC9A- 36C1D2C6DB33}");

loan.Import("C:\\Ebberson.txt", LoanImport-
  Format.FNMA3X);
```

Note that when you import a loan file into an existing Loan object, the EncompassObjects automatically preserves the identifying fields of the existing loan, specifically the Loan’s GUID, the Loan Number, and the MERS Number.

Exporting loans thru the API is done using the Loan object's Export() method. This method requires that you provide a customer-specific export key as a parameter, which is used to validate your company's Encompass license prior to permitting the export. The validation process requires that your application have Internet access and that HTTPS requests from that computer be permitted without any additional proxy authentication.

As a rule, your export key is the same as the Encompass SDK CD Key you use to activate the Encompass SDK. If you are a licensed partner and your application is intended for sale or distribution to multiple companies using Encompass, you must use your own export key when calling the Export method - do not use the export key owned by the company to whom you have sold your software. This ensures that exports done by the application are properly recognized as originating from the partner's software.

Once you have your export key, invoking the Export method is done as follows:

```
Loan loan = session.Loans.Open("("{AA6C8474-05D8-3D32-
    BC9A- 36C1D2C6DB33}";

loan.Export("C:\\Ebberson.fnm", "ABCDEFGHIJ",
    LoanExportFormat.FNMA32);
```

In the example above, the sample "ABCDEFGHIJ" export key is used.

Accessing the Loan Log

The loan log contains historical, current, and future loan-related activities and events.

The loan log contains the following types of entries.

- Conversations – Entries of e-mail and phone communications.
- Documents – Document tracking details.
- Milestones – Details including milestone completion dates and expected completion dates of future milestones.
- Milestone Tasks – Provides a record of the tasks to be completed in order to progress to the next milestone.
- EDM Transactions – Provide a record of documents sent to or requested from a borrower using the Electronic Document Management features of Encompass.
- Status Online Updates – Log updates to the Status Online web site associated with a loan when a triggering event occurs.
- Received Documents – Record the receipt of a fax or other electronic document imported via the Encompass eFolder.
- Preliminary Conditions – Details the conditions that are required to close the loan.
- Underwriting Conditions – Entries that provide the status of the conditions associated with the loan.
- Post-closing Conditions – Details for post-closing or shipping conditions associated with the loan.
- Lock Requests – The details of a rate lock request. (Banker Edition only)
- Lock Confirmations – Used to confirm an existing Lock Request. (Banker Edition only)
- Lock Denials – Provides the details for when a rate lock request is denied. (Banker Edition only)

The EncompassObjects API provides interfaces to access log entries and create your own entries in the loan log. You can use the properties and methods to implement your own user-notification system based on events specific to your business. Use the Loan object's Log property to access the loan log.

The LoanLog object has a different collection-valued property for each entry type in the log. Additionally, each entry type is implemented as its own class within the EllieMae.Encompass.BusinessObjects.Loans. Logging namespace. For example, information regarding a conversation is held in a Conversation object, and the information related to the passing of a milestone is held within a MilestoneEvent object. Each of the entry types derives from a common base class, LogEntry, which permits log entries to be handled in a generic manner when appropriate.

Conversation Log Entries

Conversation log entries track communications with customers, partners, and vendors, and provide an alert mechanism to notify users of required actions and tasks. A conversation entry can have two dates: the date on which the conversation occurred and, optionally, the date by which the loan associate must follow-up. When a conversation entry is marked for follow-up and the due date expires, the entry becomes an alert.

The following code iterates over all of the conversation log entries for the current loan.

```
Loan loan = session.Loans.Open("("{AA6C8474-05D8-3D32-
    BC9A- 36C1D2C6DB33}")

foreach (Conversation conv in loan.Log.Conversations)
    Console.WriteLine("On " + conv.Date + ", a conversation
    was held with " +conv.HeldWith);
```

The following code detects conversation log entries with alerts.

```
Loan loan = session.Loans.Open("("{AA6C8474-05D8-3D32-
    BC9A- 36C1D2C6DB33}");;

foreach (Conversation conv in loan.Log.Conversations)
    if (conv.IsAlert)
        Console.WriteLine("Follow-up for the " + conv.Date +
            " conversation with " + conv.HeldWith + " is past
                due!");
```

Milestone Event Log Entries

Loan processing tasks are divided into stages by milestones. As tasks are completed at a stage, a milestone is reached and work begins towards the next milestone.

The loan log maintains a history of when each milestone is achieved as well as a schedule of expected completion times for future milestones. Through these log entries you can modify a loan's milestone schedule, check for past-due work, and mark a milestone as being completed.

The following code uses the LoanLog and the MilestoneEvent class to determine if a loan is completed, or if not yet completed, when completion is expected.

```
Loan loan = session.Loans.Open("{AA6C8474-05D8-3D32-
  BC9A- 36C1D2C6DB33}");

Milestone completion = session.Loans.Milestones.GetItem-
  ByName("Completion");

MilestoneEvent e = loan.Log.MilestoneEvents.GetEvent-
  ForMilestone(completion);

if (e.Completed)

Console.WriteLine("The loan was completed on " +
  e.Date.ToString());

else if (e.Date != null)

Console.WriteLine("The loan is scheduled to be completed on
  " + e.Date.ToString());

else

Console.WriteLine("The loan is not yet scheduled for
  completion");
```

In the code above you must check if the date of a MilestoneEvent is null (a variant of type VT_EMPTY for COM clients) before attempting to access its value. A null date indicates that the milestone is not scheduled for completion, usually because the loan has not progressed past the *Started* milestone. Once a loan advances beyond the first milestone (other than File Started), the expected completion dates of all subsequent milestones are calculated.

Document Log Entries

Using the loan log's TrackedDocuments property, you can examine and modify document tracking entries. You can also add your own TrackedDocument instances to the log, making them visible to users who view the Document Tracking screen.

Each tracked document is associated with the milestone for which it is required and up to five dates: ordered, reordered, expected receipt, actual receipt, and expiration. Your system administrator defines document tracking rules for the expected number of days between ordering and receipt, and between receipt and expiration. If the number of days is exceeded, the document log entry becomes an alert.

The following code adds a new, custom document to the loan log, making it available for tracking through the eFolder interface. The document is associated with the loan's current milestone, ordered today, with an expected receipt date of 10 days from today.

```
// Add a document tracking entry associated with the next
  milestone
TrackedDocument customDoc = loan.Log.Tracked
  Documents.Add("My Custom Document",
  loan.Log.MilestoneEvents.NextEvent.Milestone);

// Mark the document as ordered today and due to be
  received in 10 days
customDoc.OrderDate = DateTime.Today;
customDoc.DueDays = 10;

loan.Commit();

loan.Unlock();
```

EDM Transactions

Using the Electronic Document Management (EDM) features of Encompass, loan-related documents can be sent to or requested from borrowers in a completely digital manner. Whenever a document is sent or requested, Encompass generates of log of this event and stores it along with the details of the documents involved and the Encompass user who performed the transaction.

For example, you can determine the set of all documents that have been requested from the borrower to be signed and returned as follows:

```
// Open a loan from the "My Pipeline" folder
Loan loan = session.Loans.Open("{AA6C8474-05D8-3D32-
  BC9A- 36C1D2C6DB33}");
// Iterate over all of the EDM transactions in the loan's log
  foreach (EDMTransaction txn in loan.Log.EDMTransactions)
{
  // Iterate over the list of documents looking for items // that
  are to be signed and returned by the borrower.
   foreach (EDMDocument doc in txn.Documents)
     if (doc.Action == EDMDocumentAction.SignAnd-Return)
       Console.WriteLine(doc.Title + " requested on " + txn.Date
  + " by " + txn.Creator + " as " + doc.ActionAsString);
}
// Close the loan to release its resources
loan.Close();
```

The Encompass API does not provide interfaces for requesting or sending documents, nor does it permit you to create your own EDM Transaction log entries. You can, however, modify the comments associated with a transaction and remove transaction log records. Note that removing a transaction log record will not change the state of the associated documents in the Encompass eFolder.

Received Documents

Besides the ability to send documents to and request documents from borrowers, the Encompass EDM add-on permits your organization to receive faxes back from borrowers and import them directly into Encompass. When a borrower faxes a document to the Encompass Fax Servers using the Encompass-generated fax cover sheet, the Encompass client application notifies the user of the document's arrival and provides the mechanisms to retrieve it from the Ellie Mae servers.

Whenever a fax is received through this mechanism, a ReceivedDocument log entry is created in the loan. Using this log entry you can determine the date and time the fax was received, the phone number from which it was sent, and the number of pages it included. Additionally, you can retrieve the electronic image of each page received through the ReceivedDocument's GetPages() method. Pages received through the fax import interface are automatically stored as unassigned attachments on the loan (see the heading “Attachments” on page 18). Once the fax is received, an Encompass user will generally associate the pages of the fax with one or more of the documents requested from the borrower.

Regardless of the associations made to the requested documents, the ReceivedDocument object will always maintain the entire list of the original pages received (unless you delete those pages from the loan). Because faxes can only be received through the EDM mechanism, you cannot create your own ReceivedDocument log entries. You may only modify the comments associated with the entry or delete the entry altogether. Note that deleting a ReceivedDocument log entry will not cause the attachments representing the pages of the fax to be removed from the loan.

Attachments

As part of the emphasis Encompass puts on storing and submitting electronic documents, the API provides methods to add, remove, and extract digital attachments to a loan. Within the Client Application, these attachments show up as part of the eFolder and within the Document Management interface. Using the EncompassObjects, you can directly insert documents into Encompass from a custom data source, e.g., your own automated scanning department.

All attachment-related functionality on a loan is accessed through the loan's Attachments property. Through this interface you can create a new attachment, delete an attachment, iterate over the existing attachments, or even extract an attachment to disk. The following demonstrates attaching a TIFF image to an existing loan.

```
// Open the session to the remote server
Session session = new Session();
session.StartOffline("myserver", "mary", "maryspwd");
// Open the loan using the GUID specified on the command
  line
Loan loan = session.Loans.Open("{AA6C8474-05D8-3D32-
  BC9A- 36C1D2C6DB33}");
loan.Lock();
// Create a new attachment by importing it from a TIFF
  document on disk
Attachment att = loan.Attachments.Add("C:\\Scanner
  Output\\MyAppraisal.tif");
// Save the changes to the loan, which commits the new
  attachment
loan.Commit();
loan.Close();
```

Note that just as for changes to the loan's fields, you must lock a loan prior to adding an attachment. Similarly, you must invoke the loan's Commit() method to save all changes made to the Attachments collection, otherwise they will be lost. Unlike the loan fields, a loan must be committed prior to attempting to add an attachment.

Currently, Encompass supports attachments only of the following types:

- Text documents (*.txt)
- Adobe PDF documents (*.pdf)
- Microsoft Word Documents (*.doc)
- TIFF images (*.tif)
- JPEG images (*.jpg, *.jpeg, *.jpe)

When importing an attachment, it must have an extension matching one of the document types listed above.

Once an attachment is imported into a loan, the next processing step is usually to associate that attachment with one of the TrackedDocument objects in the loan's Log. For example, if you import the scanned image of an appraisal, you should locate the Appraisal document within the log and attach the image to it. Each attachment can be associated with at most one document. Attachments which are not associated with any document will appear as "Unassigned" in the Document Manager.

The following code demonstrates linking an attachment to an existing TrackedDocument record.

```
// Open and lock the loan on which the attachment will be
  made
Loan loan = session.Loans.Open("{AA6C8474-05D8-3D32-
  BC9A- 36C1D2C6DB33}");
loan.Lock();
// Create the new attachment by importing it from a TIFF
  image
Attachment att = loan.Attachments.Add("C:\\Scanner
  Output\\MyAppraisal.tif");
// Now attach the new Attachment to the Appraisal on the
  loan
LogEntryList appraisals = loan.Log.Tracked-
  Documents.GetDocumentsByTitle("Appraisal", false);
if (appraisals.Count > 0)
{
   TrackedDocument appraisal = (TrackedDocument)
     appraisals[0];
   appraisal.Attach(att);
}
// Save the changes to the loan, which commits the new
  attachment
loan.Commit();
loan.Close();
```

Once an association is made (either through the API or the client application), you can retrieve the set of Attachments associated with a document through the TrackedDocument's GetAttachments() method. Note that if a TrackedDocument is deleted that had assigned to it one or more attachments, those attachments will automatically revert to being unassigned (i.e. no longer associated with any document). Conversely, if an attachment is deleted from the loan, any existing associations with documents will be broken.

Loan Templates

Encompass allows users to create publicly- or privately-available loan templates to facilitate the creation of loans that have common characteristics or data. By using templates, the data entry requirements and the accompanying likelihood of error are greatly reduced when boilerplate data is needed.

The API provides a limited number of objects and methods for accessing loan templates created through the Encompass application, and for using them to create new loan instances. The relevant classes are contained in the EllieMae.Encompass.BusinessObjects. Loans.Templates namespace and access to these objects is made through the Loans.Templates property.

The primary requirement for using a Loan Template is that your code is able to successfully retrieve the template from the Encompass Server. Every template, regardless of its type, resides in a file system-like hierarchy of folders. And, just as with the Windows file system, each template has a path specification which is used to access the object.

Path specifications for templates use the following format:

<domain>:\<folder>\<subfolder>\...\<templatename>

The <domain> for a template is determined by whether it is publicly-accessible (in which case the “public” domain is used) or only accessible by the currently logged in user (the “personal” domain). The remaining portions of the path specification are common to the Windows file system, except that Encompass permits any printable character to be included in the folder and template names, except for the path delimiter (the back slash character).

For example, the path specification “public:\MyBank Templates\ARMs\5 – 1 ARM” would retrieve the template named “5-1 ARM” from the “ARMs” subfolder of the “MyBank Templates” folder within the public domain.

Using the above path specifications, you can retrieve Loan Templates from an Encompass Session and use it to instantiate a new Loan object:

```
LoanTemplate template = (LoanTemplate)
  session.Loans.Templates.GetTemplate(TemplateType.
  LoanTemplate,
  "public:\MyBank Templates\ARMs\5 – 1 ARM");
Loan loan = template.CreateNew();
```

The EncompassObjects API provides access to all of the different template types supported by Encompass:

- Loan Templates
- Closing Cost Templates
- Loan Programs
- Document Sets
- Data Templates
- Input Form Sets
- Task Sets

Each template type is represented by its own object type which derives from the common Template base class. When you retrieve a template from the server, you can cast it to the appropriate object type to access functionality specific to that template type.

The following code retrieves a DataTemplate and checks the value of field 315 (Broker Company Name) within the template.

```
string path = @"public:\My Company Templates\California
  Data Template";
DataTemplate template =
  (DataTemplate)session.Loans.Templates.GetTemplate-
  (TemplateType.DataTemplate, path);
string brokerName = template.Fields["315"].FormattedValue;
```

A template can be applied to an existing Loan object by using the Loan’s ApplyTemplate() method.

```
loan.ApplyTemplate(template, true);
```

The second parameter to the ApplyTemplate method indicates whether the data from the template should be “appended” to the loan instead of overwriting what’s in the loan. If the parameter’s value is true, only non-empty values from the template will be applied to the loan. Thus, if a field is populated in the loan and unpopulated in the template, the field's value will be preserved. If the parameter is false, all fields from the template, including blank fields values, will overwrite the data in the loan.

The API also provides methods for examining and navigating the directory hierarchy in which the templates are stored. This functionality can be used to provide a user with a tree-type view of the templates available in Encompass.

The primary method required for traversing the template directory structure is the Templates.GetTemplateFolder-Contents() method, which provides a listing of all templates and subfolders contained in a specific folder in the hierarchy. Each entry in the folder’s contents list is represented by a TemplateEntry object. The TemplateEntry’s EntryType property indicates whether the entry represents a subfolder or an actual Template object.

To begin navigating the directory tree, you must first obtain a reference to the root folder element. For the “public domain” hierarchy, the root is a fixed value represented by

TemplateEntry.PublicRoot. For example, the following code enumerates the subfolders and templates within the root directory of the public domain of Loan Program templates.

```
TemplateEntryList entries =
  session.Loans.Templates.Get-TemplateFolderContents-
  (TemplateType.LoanProgram, TemplateEntry.PublicRoot);
foreach (TemplateEntry entry in entries)
{
  if (entry.EntryType == TemplateEntryType.Folder)
    Console.WriteLine("Subfolder: " + entry.Name);
  else
    Console.WriteLine("Template: " + entry.Name);
}
```

Within a user's private domain, the root folder is accessed using the TemplateEntry.GetPersonalRoot() method.

```
TemplateEntry peroot = TemplateEntry.GetPersonalRoot-
  ("jsmith");
```

Using the TemplateEntry object and the Templates.Get-TemplateFolderContents() method, you can recurse through the entire folder hierarchy. Alternatively, if you know the path of a particular folder and need to retrieve its contents, you can use the TemplateEntry.Parse() method to convert a string to the corresponding TemplateEntry object.

```
string path = @"public:\My Company Templates\California
  Data Template";
TemplateEntry e = TemplateEntry.Parse(path);
TemplateEntryList entries =
  session.Loans.Templates.GetTemplateFolderContents-
  (TemplateType.LoanProgram, e);
```

Access Rights

Encompass controls access to individual loans using both implicitly and explicitly assigned access rights. Implicit access rights are those determined based on the user's assignment as a loan team member, a user's persona, or a user's place within the organization hierarchy. These rights are automatically granted and cannot be revoked. For example, an administrator at the top level of the organization hierarchy will have full access to every loan in the system, regardless of their explicitly assigned rights.

Assigned rights are those granted to a user for a specific loan and which can later be revoked. There are three levels of assigned rights for a loan: no access, read/write access and full access. No access, which is the default, means that the user has no additional rights to the loan beyond those granted by their implicit rights. Granting a user read/write access to a loan allows that person to open the loan, make modifications and save it. A user with full access can both modify a loan and can grant access rights to or revoke access rights from other users.

When assigned rights and implicit rights are combined, the result is a user's effective rights. A user's effective rights to a loan are the greater of their implicit rights and their assigned rights. Thus, assigned rights can be used to grant loan access to a user who would otherwise not be permitted to access the loan, but cannot be used to prevent access to a user with implicit rights to the loan. Note that it is possible for a user to have read-only rights to a loan, but this access level can only be obtained through implicit rights, such as a supervisor who has been granted read-only access to his subordinate's loans in the user setup screen.

The API includes functions for determining a user's assigned and effective rights for a particular loan as well as the ability to modify a user's assigned rights. These functions require that the logged-in user have Full (effective) rights to a loan in order to access or modify a user's assigned rights.

The following code demonstrates granting a user Full access rights to a loan.

```
User user = session.Users.GetUser("jsmith");
Loan loan = session.Loans.Open("{AA6C8474-05D8-3D32-
  BC9A- 36C1D2C6DB33}");
if (loan.GetEffectiveAccessRights(user) < LoanAccess-
  Rights.Full)
  loan.AssignRights(user, LoanAccessRights.Full);
```

Unlike changes to a loan's Fields collection, changes to a loan's access rights occur immediately without requiring a call to the Commit method. Additionally, you cannot assign rights to a loan until it has been committed for the first time.

The API also provides a function for retrieving a list of all users who have assigned rights for the loan. The following example demonstrates how to display all users who have assigned rights, along with their assigned and effective rights.

```
foreach (User user in loan.GetUsersWithAssignedRights())
  Console.WriteLine("User '" + user + "' has assigned rights '" +
    loan.GetAssignedAccessRights(user)
    + "' and effective rights '"
    + loan.GetEffectiveAccessRights(user) + "'");
```

The Loan object provides one rights-related additional method which any user can invoke: GetAccessRights(). This method returns the current user's effective rights to the current loan. This information can be used to determine whether operations such as assigning rights to another user will succeed or cause an error.

Roles and Loan Associates

Every Encompass system defines the *roles* which users will take on within a loan as it moves from milestone to milestone. Typical examples include the Loan Officer, Loan Processor and Underwriter roles. Roles can be used to determine the set of business rules which govern the user's access to specific loan features or areas.

Users are assigned to roles on a loan-by-loan basis. Thus, a user who is in the Loan Officer role for one loan may be in the Loan Processor role for another. Additionally a user can

be assigned to multiple roles within the same loan file, for example both Loan Officer and Loan Processor. Whenever a user is assigned to a role within a loan, that user is said to be a *loan associate* for the loan.

The EncompassObjects API allows you to inspect and modify the set of loan associates assigned to a particular loan through the Loan.Associates collection. For example, you can determine what, if any, users are assigned to the Loan Officer role.

```
Role lo = session.Loans.Roles.GetRoleByAbbrev("LO");
LoanAssociateList associates = loan.Associates.GetAssociates-
  ByRole(lo);
foreach (LoanAssociate associate in associates)
  Console.WriteLine(associate.User.FullName);
```

Note that a user's assigned role within a loan is different from their loan access rights – a user who can access the loan may not be a loan associate, meaning they have no assigned role within the loan. On the other hand, a loan team member always has implicit read/write rights for a loan. These rights cannot be revoked except by removing the user as a loan team member.

Within a loan file there are two types of roles:

- Milestone roles, which are those associated with a particular milestone in the loan, and
- Milestone-free roles, which are those that are not associated with any milestone.

The association between milestones and roles is configurable for each Encompass system (Banker Edition only), and a single role can be associated with multiple milestones (although a milestone can only be mapped to a single role). Thus, multiple users can be assigned to the same role but at different milestones.

For example, the user "john" may be the loan associate assigned to the Loan Officer role for the Approval milestone while "mary" may be the Loan Officer assigned for the Funding milestone. When a role is associated with multiple milestones it is possible to have more than one loan associate in the role. However, in all other cases, Encompass limits the loan to having a single user in any given role.

You can determine whether a loan associate's role is a milestone role by accessing the LoanAssociate.MilestoneEvent property. This property will be null if the role is a milestone-free role or it will reference the MilestoneEvent log entry with which the role is bound. Similarly, if you have a reference to a MilestoneEvent object, you can directly access the LoanAssociate for that milestone. If the milestone is associated with a role, this property will be non-null and you can use it to set the user assigned to corresponding role.

```
Milestone approval =
  session.Loans.Milestones.GetItemByName("Approval");
MilestoneEvent ms = loan.Log.MilestoneEvents.GetEvent-
  ForMilestone(approval.Name);
  ms.LoanAssociate.User = session.Users.GetUserByID
  ("mary");
```

Although most roles in a loan will be assigned to a single user, Encompass allows you to assign an entire UserGroup to a role. By assigning a group to a role, every member of that group is considered to be a loan team member. Thus, every group member will have read/write access to the loan and will be permitted to perform the duties assigned to this role.

To assign a UserGroup to a LoanAssociate object, use the UserGroup property or invoke the AssignUserGroup() method on the LoanAssociates object.

```
Role shipper = session.Loans.Roles.GetRole-
  ByName("Shipper");
UserGroup shipperGroup = session.Users.Groups.GetGroup-
  ByName("Shippers");
loan.Associates.AssignUserGroup(shipper, shipperGroup);
```

The LoanAssociate object's AssignUserGroup method (and its companion method, AssignUser) will assign the specified group (or user) to any LoanAssociate object tied to the given role. If the role is associated with multiple milestones, all of the corresponding MilestoneEvents will be assigned to that group/user. If there is no milestone tied to the specified role, a milestone-free role association will be created.

Although Encompass Banker Edition allows for the customization of Roles to suit your business' needs, it is standard practice in most organizations to employ a certain set of "fixed" roles, such as the Loan Officer and Loan Processor. These roles exist in virtually every organization and have pre-defined meanings within the loan origination process.

To account for this commonality, Encompass defines four "fixed" roles:

- Loan Officer
- Loan Processor
- Loan Closer
- Underwriter

The Encompass configuration settings allow you to map your system-specific roles to these four fixed roles. For example, your administrator may have defined a role named "Loan Liason" which has been mapped to the fixed "Loan Officer" role.

In order to implement functionality which is system-independent, you can use the fixed roles to assign or retrieve loan associate information in place of using the system-specific Role values. For example, when assigning a user to a role, you can specify the fixed Loan Officer role in place of the actual underlying Role.

```
User joyce = session.Users.GetUserByID("joyce");
loan.Associates.AssignUser(FixedRole.LoanOfficer, joyce);
```

Using our example from above, this code would assign the user "joyce" to the Loan Liason role because ithas been mapped to the fixed Loan Officer role.

Using the fixed roles is completely optional and is most convenient when used with the Broker Edition of Encompass. Because the Broker Edition does not permit creation or customization of roles, the FixedRole enumeration values map directly to the Role objects of the same name. Thus, the FixedRole can be used as a convenient shortcut in place of retrieving the underlying Role object.

Although the set of roles can be different from system to system, Encompass defines one static role: File Starter. This role identifies the user who initially created the loan file. This role is always associated with the File Started milestone and cannot be associated with any other milestone. You can access this predefined role using the Loans.Roles.FileStarter property.

Lock Requests

Built into Encompass Banker Edition is a robust process by which your loan processing team can generate rate lock requests which are then either approved or denied by a separate team of users in your organization, such as your lock desk personnel. Typically, a loan processor requests a lock by completing the Lock Request Form from the Encompass Tools list. That request is then reviewed by a lock desk user using the Secondary Registration tool, which provides a rich interface for reviewing the loan's key data, populating buy-side and sell-side pricing information, and ultimately for approving or denying the rate lock.

The EncompassObjects API exposes this complete process to applications, allowing you to request, review and confirm or deny rate lock requests in your code. The following example demonstrates how to create a new lock request on an existing loan.

```
// Populate the requested lock date and lock period
loan.Fields["2089"].Value = DateTime.Today;
loan.Fields["2090"].Value = 30;

// Now create the new LockRequest. This is equivalent to
// pressing the Request Lock button on the
// LockRequestForm.
LockRequest request = loan.Log.LockRequests.Add();
```

Because a single loan can support multiple lock requests, the data associated with each request must be stored separately. When a LockRequest object is created, Encompass generates a *lock request snapshot* which is attached to the request. The snapshot consists of two sets of data:

- A read-only copy of the fields found on the Lock Request form. These fields are informational and allow the user reviewing the lock request to see the key data values on which the request is based.
- The secondary registration data, such as the buy-side and sell-side pricing data, the investor information, etc.

To access the fields in a lock request snapshot, you must use the LockRequest's Fields collection. This collection works in exactly the same way as the Loan's Fields collection but contains only the subset of fields which are part of the snapshot. The following code demonstrates how to retrieve and set the values in the snapshot for the current lock request.

```
// Retrieve the current lock request from the loan
LockRequest req = loan.Log.LockRequests.GetCurrent();

// Set the Buy side lock date and period
req.Fields["2149"].Value = DateTime.Today;
req.Fields["2150"].Value = 45;

// Set the buy-side base rate and adjustments
req.Fields["2152"].Value = 6.75;
req.Fields["2153"].Value = "45 Day Lock Period";
req.Fields["2154"].Value = 0.125;

// Set the buy-side base price and adjustments
req.Fields["2161"].Value = 99.75;
req.Fields["2162"].Value = "FICO >= 720";
req.Fields["2163"].Value = 0.25;

// Commit the changes to the snapshot
req.Fields.CommitChanges();
```

The Field IDs used in the example above are taken from the Secondary Registration form in Encompass. Note that any time you modify the values in a lock request snapshot you must invoke the CommitChanges() method to write your changes back into the snapshot. As usual, none of your changes are saved to the Encompass Server until you invoke the Loan object's Commit() method.

Besides its use above for creating new lock requests, the LoanLog's LockRequests collection allows you to access current or past lock requests. The GetCurrent() method provides quick access to the "active" lock request. You can also iterate over the collection of lock requests and use the LockRequest's Status property to determine the state of each request.

```
// Loop over the lock request collection to find any currently
// pending requests
foreach (LockRequest req in loan.Log.LockRequests)
    if (req.Status == LockRequestStatus.Pending)
        Console.WriteLine("Active lock request made on " +
        req.Date);
```

The final step in managing lock requests is either confirming or denying the lock request. The LockRequest object provides three methods which handle these operations:

LockRequest.Lock(): Marks the current request as locked but does not write the lock data back to the loan file. Additionally, the loan officer/processor receives no confirmation of the rate lock. This method is used to perform locks which are visible only to the lock desk user. These locks must still eventually be confirmed or denied using the methods below.

LockRequest.Confirm(): Confirms the rate lock and copies the buy-side rate information into the core loan file fields. This method additionally performs all the functions of the Lock() method above if Lock() was not previously called.

LockRequest.Deny(): Denies the lock request and sends notification of the denial to the loan processor.

The following example demonstrates the process for confirming an existing rate lock request.

```
// Retrieve the current lock request from the loan
LockRequest req = loan.Log.LockRequests.GetCurrent();

// Confirm the request
LockConfirmation conf = req.Confirm();
```

When you invoke the Confirm() method, a new LockConfirmation object is generated. This object can be used to quickly access certain key properties of the confirmation, such as the buy- and sell-side expiration dates and the identity of the user who confirmed the rate lock. To retrieve an existing LockConfirmation for a confirmed lock, you can use the LockRequest.Confirmation property.

Loan Servicing

If your company provides loan servicing on an interim basis after a loan is closed, the Encompass Banker Edition provides a streamlined interface for tracking payments, printing statements and managing disbursements of funds from escrow. Because you may wish to automate these processes or integrate Encompass with your external accounting system, the EncompassObjects API provides the necessary classes and methods to manipulate the servicing data for a loan.

When a loan is closed and you need to provide servicing, the first required step is to initiate the servicing process on the loan.

```
// Initiate servicing on the loan
if (!loan.Servicing.IsStarted())
    loan.Servicing.Start();
```

When servicing is started, Encompass computes the payment and escrow disbursement schedules and unlocks the fields related to servicing the loan. You can retrieve the payment schedule using the LoanServicing.GetPaymentSchedule() method.

```
// Retrieve the payment schedule
PaymentSchedule sched = loan.Servicing.GetPayment-
  Schedule();

foreach (ScheduledPayment pmt in sched.Payments)
    Console.WriteLine("A payment of $" + pmt.Total +
    " is due on " + pmt.DueDate);
```

Whenever a payment is received or a disbursement is made, a record is recorded in the servicing history of the loan as a transaction. Encompass supports the following different transaction types:

- Payment
- Payment Reversal
- Escrow Disbursement
- Escrow Disbursement Reversal
- Escrow Interest Payment
- Other

All of the above transactions types are accessed, added or removed through the LoanServicing object's Transactions property. In the EncompassObjects API, each transaction is represented by an instance of a class that derives from the abstract ServicingTransaction class, found in the EllieMae.Encompass.BusinessObjects.Loans.Servicing namespace. For example, a payment transaction is represented by the Payment class, and a new payment can be recorded as shown below.

```
// Add a new payment to the servicing history of the loan
Payment pmt = loan.Servicing.Transac-
  tions.AddPayment(DateTime.Now);

// Set the properties of the payment to match what was
// received
pmt.Principal = 700;
pmt.Interest = 1500;
pmt.Escrow = 250;
pmt.AdditionalPrincipal = 150;

// Indicate the manner in which payment was made
pmt.PaymentReceivedDate = DateTime.Parse("6/5/2008");
pmt.PaymentMethod = ServicingPaymentMethod.Automat-
  edClearingHouse;
pmt.AccountHolder = "Janet T. Barnes";
pmt.AccountNumber = "10339442-4";
pmt.InstitutionName = "First Federated Bank";
pmt.InstitutionRouting = "444444444";
pmt.Reference = "AB3-4434-223";
```

When a payment is added, the interim servicing engine will apply the payment to the next scheduled statement and adjust the payment schedule as required.

Reversing a previously created payment is done by invoking the Payment object's Reverse() method. A similar technique is available for creating and reversing an escrow disbursement transaction.

```
// Retrieve the most recent payment from the servicing history
ServicingTransactionList payments = loan.Servicing.Transac-
  tions.GetTransactions(ServicingTransactionType.Payment);
Payment lastPayment = (Payment) payments[payments.Count
  - 1];

// Reverse the payment, which will re-apply the principal

// back to the loan balance.
lastPayment.Reverse(DateTime.Now);
```

The example above also demonstrates how to retrieve past transactions from the servicing history of the loan. The LoanServicingTransactions.GetTransactions() method will retrieve all transactions of a specified type using the ServicingTransactionType enumeration provided by the API.

Batch Loan Update

The EncompassObjects API's Batch Loan Update feature allows you to quickly update multiple loans without having to individually open, lock and save each loan file. The set of loans to be updated can either be defined explicitly by providing a list of the individual loans' GUIDs or by using the QueryCriterion objects described in the Reports and Queries section of this document. For example, you can direct the API to update all loans where the subject property resides in a particular state.

Although the batch update feature is very efficient for performing a bulk update of large numbers of loans, this efficiency comes at a cost. In particular, when you use the Batch Loan Update, the following rules apply:

- All business rules are ignored.
- Calculations are not performed, regardless of whether they are standard Encompass calculations or custom field calculations.
- Only field values with explicit field IDs can be updated – Loan Log data cannot be modified using the Batch Loan Update.
- If a user is currently in a loan and has it locked, the batch loan update will be applied on top of any changes they make. Thus, if the user modifies a field which is also modified in the batch loan update, the user's value will be overwritten.
- Audit trail data will be updated if the fields being modified are being audited.
- The Reporting Database will be updated and reflect the correct values.
- Pre-defined index data is not updated for the loans, meaning inaccurate data may be displayed in the pipeline or reports. For example, if you update the Loan Amount (field 1109) and the user includes the built-in Loan Amount column in the pipeline, the value displayed will reflect the old value until the loan is opened and re-saved or until a Rebuild Pipeline operation is performed. To avoid this problem use fields from the Reporting Database in the pipeline and reports instead of the pre-defined fields provided by Encompass.

Because the Batch Loan Update has the ability to circumvent business rules, the API restricts this functionality to users with the Administrator or Super Administrator persona. Attempting to perform an update without the necessary rights will result in an exception.

More significantly, because calculations are not executed during the batch update process, it is recommended that you do not use the batch update to modify fields that are used within any core or custom calculations. For example, using the Batch Loan Update to modify a loan's interest rate or loan amount would be inappropriate since those fields have widespread impact on other calculated field values in the loan. Instead, the feature is intended to be used to update items such as the Broker's address or contact information.

To perform a batch update in your code, start by creating a new instance of the BatchUpdate class. The class provides three overloads for its constructor: one which allows for an explicit list of loan GUIDs, one which allows for a QueryCriterion object and one which permits only a single loan GUID. The following example constructs a BatchUpdate to modify all loans where the subject property resides in California.

```
StringFieldCriterion cri = new StringFieldCriterion();
cri.FieldName = "Loan.State";
cri.Value = "CA";
BatchUpdate batch = new BatchUpdate(cri);
```

Once constructed, you must specify the loan field/value pairs which are to be updated. Each batch can be used to update multiple fields; however, the set of fields and their values must be the same for each loan being updated. The following code adds fields to the batch update to set the broker company information on the 1003 form.

```
batch.Fields.Add("315", "My Mortgage Loan Company");
batch.Fields.Add("319", "22 Main Street");
batch.Fields.Add("313", "Arlington");
batch.Fields.Add("321", "VA");
batch.Fields.Add("323", "22222");
```

Once you have added the value for one or more fields to the batch, you can submit the batch to the Encompass Server.

```
session.Loans.SubmitBatchUpdate(batch);
```

The call to SubmitBatchUpdate() should return very quickly, although the time taken may be somewhat longer if the fields being modified are in the Reporting Database or are being audited. The update of the loans is synchronous, meaning that once the method returns, the update is complete and the loans should immediately reflect the changes submitted in the batch.

Section 5

Managing Contacts

The EncompassObjects API provides a set of classes to access and update the contacts database. The contacts database is split into two types: one type maintains the personal information of past, present, and potential borrowers and one type maintains rolodex-type information for business contacts and partners. With only a few exceptions, these two types of contacts are treated and represented identically within the API.

Retrieving, Modifying, and Saving Contacts

The logical structure of the classes used for fetching and saving contacts is roughly equivalent to that used by the Loan-related portion of the API. Off the Session object exists a Contacts property, the entry-point into all of the contact-related functionality. Using this property you can retrieve a contact from the server, assuming you know the contact's unique ID and type (borrower or business contact).

```
Contact contact = session.Contacts.Open(102,
  ContactType.Borrower);
```

The same method is used to retrieve borrower contacts and business contacts. What is returned from the method, however, is not just a Contact instance, but an object that derives from the Contact class: BorrowerContact if the contact retrieved is a borrower, or BizContact if the contact is a business partner or contact. The following code accesses the full functionality of the returned object.

```
BorrowerContact contact = (BorrowerContact)
  session.Contacts.Open(120, ContactType.Borrower);
```

The common base class, Contact, contains all of the properties and methods shared by both borrowers and business contacts. Often it may be sufficient to deal with the object without ever casting it to its derived type. For example, the following code retrieves a contact, modifies its address, and saves it back to the server.

```
Contact contact = session.Contacts.Open(120,
  ContactType.Borrower);
contact.BizAddress.Street1 = "202 Fillmore St.";
contact.BizAddress.City = "Arlington";
contact.BizAddress.State = "VA";
contact.Commit();
```

As with the Loan object discussed previously, any changes made to the Contact object are not saved to the database until you invoke the object's Commit() method. Unlike the Loan object, the Contact object does not support methods to Lock() and Unlock() the contact. When the Commit method is called, all of the contact's data is transferred back to the server and updated. Thus, if another user modified the contact between the time it was retrieved and the time Commit() was called, those changes would be overwritten by the current process.

Another feature of contacts that is different from the Loan object is that all of its fields are distinct, named properties instead of having string-based Field IDs. As a result, inspecting the list of available properties and methods on the Contact object and its derived classes defines the features available for the client application.

Creating Contacts

Creating a new Contact is a simple operation. The only required data is the type of contact (borrower or business) to be created:

```
Contact newContact =
  session.Contacts.CreateNew(ContactType.Biz);
newContact.FirstName = "Margaret";
newContact.LastName = "Davis";
newContact.Commit();
```

As with the loan objects, calling CreateNew() simply creates an empty shell for a new contact. The call to Commit() actually saves the new Contact to the contacts database.

Custom Fields

The administrator can define up to 100 custom fields for each type of contact (borrower and business). These fields store custom data for the borrower and business contacts stored in the database.

The EncompassObjects API includes properties and methods for accessing, modifying, and performing queries against the values stored in the custom fields.

Accessing Contact Custom Field Data

Accessing the custom field data is done through the CustomFields property of the Contact object.

```
Contact contact = session.Contacts.Open(355,
  ContactType.Biz);
String color = contact.CustomFields.["Favorite Color"].Value;
```

Changing Contact Custom Field Data

Changes to custom field values are saved when the Commit() method is called on the parent Contact object.

```
Contact contact = session.Contacts.Open(355,
  ContactType.Biz);
contact.CustomFields["Favorite Color"].Value = "Blue";
contact.Commit();
```

Working with Field Types

Each custom field is associated with a field type (NONE indicates no selection was made by the user and the associated field value is treated as s string). The field type indicates the range and format of values the custom field can store. For example, a custom field defined as a ZIPCODE field can store up to 10 characters in the format XXXXX-XXXX. When you set the value of a field through the API, the EncompassObjects automatically attempts to convert the provided value into a valid value for the associated field type. For example, the value 123456789 is reformatted to 12345-6789 if set as the value of a ZIPCODE field type.

Some input values cannot be reformatted in a meaningful way. For example, the value ABC cannot be converted into a numeric ZIPCODE. Attempts to set a custom field to a value that cannot be converted to a meaningful value based on the field's type results in an exception. The EncompassObjects API exposes a property on the ContactCustomField object that allows the developer to detect the field type so that potential format or content errors can be identified and avoided prior to setting the field's value.

For the DROPDOWN-EDITABLE or DROPDOWN field types, the administrator typically defines a list of potential values that the custom field can assume. The API exposes this list through the PossibleValues property of the ContactCustomField object. In the case of the DROPDOWN field type, attempting to set the field to any value that does not exactly match one of the values in the PossibleValues list results in an exception. The DROPDOWN-EDITABLE field type permits the assignment of any value, whether or not it is in the list of PossibleValues.

You can also use custom fields to perform queries against the Contacts database. For more information about using Custom Fields in queries, see “Contact Custom Field Queries” on page 34.

Linking Contacts to Loans

Because Contacts and Loans can be created and modified separately, they can largely be viewed as independent of one another. However, because many of the contacts in your system represent the borrowers and business partners with whom you are transacting business, a logical connection exists between the two sets of data.

Encompass provides a mechanism for formalizing the relationships between your contacts and your loans. By tying individual contacts to their respective loans, you gain the ability to look at historical data, search out contacts based on loan attributes and avoid duplication of work.

Encompass allows for a many-to-many mapping between the contacts in your system and the loans. Each mapping is made more precise by requiring you to specify the relationship that the contact fulfills within the loan, such as "appraiser" or "borrower." A single contact may generally fulfill multiple relationships within a loan (e.g. "title company" and "notary") while each relationship may only be mapped to a single contact.

More precisely, a loan breaks its relationships down into two categories: Borrower Relationships and Business Relationships. A borrower relationship exists for each borrower and coborrower within the loan file. For example, a loan that has two "borrower pairs" will have four borrower relationships that can be assigned. Each borrower relationship can be assigned to a different Borrower Contact - a contact cannot fulfill multiple borrower relationships on the same loan.

The business relationships within a loan are a fixed set, but a single business contact can be assigned to multiple business relationships. However, a given relationship, such as "Appraiser," can only be assigned to a single business contact. Assigning a new contact to that relationship will remove any prior contact that was assigned.

Linking and unlinking a contact to a loan is done through the Loan object's Contacts collection. The following demonstrates how a Borrower Contact is linked to the primary Borrower within a Loan.

```
// Fetch the BorrowerContact record
BorrowerContact contact = (BorrowerContact)
  session.Contacts.Open(…,
      ContactType.Borrower);

// Link the contact to the primary Borrower from the Loan
loan.Lock();
Borrower borrower = loan.BorrowerPairs[0].Borrower;
loan.Contacts.LinkToBorrowerContact(contact, borrower);
loan.Commit();
```

Note that in order to create the link, the loan must be locked. The link is not saved until you invoke the Loan's Commit() method.Linking a loan to a Business Contact is done by specifying the relationship to which the contact is to be assigned. In the code below, the contact is assigned as the Appraiser on the open loan.

```
// Fetch the BizContact record
BizContact contact = (BizContact) session.Contacts.Open(…,
      ContactType.Biz);

// Link the contact as the Appraiser of the loan
loan.Lock();
loan.Contacts.LinkToBizContact(contact, LoanContactRela-
  tionshipType.Appraiser);
loan.Commit();
```

You can also use the Loan's Contacts collection to retrieve the set of contacts already linked to the existing loan. The individual elements of the Contacts collection are represented by the LoanContactRelationship object, which provides the identifying information for the contact as well as the type of relationship to which it is assigned.

```
// Display the list of all linked contacts for a loan
foreach (LoanContactRelationship rel in loan.Contacts)
{
   Contact c = rel.OpenContact();
   Console.WriteLine(c.FullName + " is the " + rel.Relation-
  shipType);
}
```

Similarly, you can access a Contact's set of linked loans by calling the GetLoanRelationships() method.

```
// Display the list of all linked loans for a Contact
foreach (LoanContactRelationship rel in contact.GetLoanRela-
  tionships())
{
   Loan loan = rel.OpenLoan();
   Console.WriteLine("Contacts is the " + rel.Relationship
         + " in loan " + loan.LoanNumber);
}
```

Another important aspect of linking contacts to loans is the ability to perform queries on the Contact records using data from their linked loans as part of the filter criteria. For example, it is possible to retrieve all BorrowerContact records that are linked to loans for greater than $300,000. For more information on running loan-related contact queries, see the Reports and Queries section below.

Section 6

Reports and Queries

A common method to retrieve loan and contact data from the Encompass server is to perform an ad hoc query against the set of objects saved on the server. The EncompassObjects API provides several classes and a pair of methods to make performing queries relatively easy without exposing you to the underlying query language and table structure of the server.

Criterion Classes

Whether you are querying for Loans or Contacts, the basic query mechanism is the same. The EllieMae.Encompass.Query namespace provides a set of classes to define and build the query criteria that the server will execute. The primary query classes are:

- StringFieldCriterion, used to specify a query against a field whose values are represented as strings.
- NumericFieldCriterion, used to specify a query against a field whose values are represented as numeric values, such as integer, floating point, and currency.
- DateFieldCriterion, used to specify a query against a field whose value is a date or partial date.

Think of a query in the EncompassObjects API as a set of criterion objects joined by logical AND and OR operations to create an arbitrarily complex query. Each criterion object contains the information required to know how a single database field is involved in the query. For example, the following code queries for all Borrower Contacts that have the last name “Smith”.

```
StringFieldCriterion cri = new StringFieldCriterion();
cri.FieldName = "Contact.LastName";
cri.Value = "Smith";
ContactList contacts = session.Contacts.Query(cri, Contact-
   LoanMatchType.None, ContactType.Borrower);
```

In this simple example, the only pieces of data provided were the field name, Contact.LastName, and the value to be found (the parameters passed to the Query() method are discussed below). The StringFieldCriterion object provides even greater flexibility. For example, you can expand the query by searching for any borrower whose last name starts with "S":

```
StringFieldCriterion cri = new StringFieldCriterion();
cri.FieldName = "Contact.LastName";
cri.Value = "S";
cri.MatchType = StringFieldMatchType.StartsWith;
ContactList contacts = session.Contacts.Query(cri, Contact-
   LoanMatchType.None, ContactType.Borrower);
```

In general, you can use each of the three criterion classes to specify the canonical name of the field to query, the value to query, and the means by which the database value will be compared to the specified value. For example, the following code queries for all business contacts with fees less than $350.

```
NumericFieldCriterion cri = new NumericFieldCriterion();
cri.FieldName = "Contact.Fees";
cri.Value = 350.00;
cri.MatchType = OrdinalFieldMatchType.LessThan;
ContactList contacts = session.Contacts.Query(cri, Contact-
   LoanMatchType.None, ContactType.Biz);
```

Note that the potential values for the MatchType property on the NumericFieldCriterion object differ from those of the StringFieldCriterion object’s MatchType property.

The DateFieldCriterion object adds another property to control the precision with which the database value is compared to the specified test value. This permits you to compare only the year, only the month and year, or only the month and day of the database value against the value specified in the criterion. For example, in the following code, the Recurring precision specified indicates that only the month and day should be compared. The result is a match for any borrower whose birthday falls on March 15 (regardless of the year).

```
DateFieldCriterion cri = new DateFieldCriterion();
cri.FieldName = "Contact.Birthdate";
cri.Value = DateTime.Parse("3/15/2000");
cri.Precision = DateFieldMatchPrecision.Recurring;
```

Logical Operations

Once you can create single-field queries using the three base criterion classes, the next step is to join these criteria into complex logical constructs. Each of the three criterion classes defined above is derived from a common, abstract base class named QueryCriterion. This class provides two methods to each of the criterion objects:

- And() – joins the current criterion with another criterion using AND logic to generate a new criterion object.
- Or() – joins the current criterion with another criterion using OR logic to generate a new criterion object.

The result of both operations is a new object of the type QueryCriterion. You can use the output of these operations as input into further And() or Or() logic to create arbitrarily complex queries. For example, the following query locates all loans for properties in Arizona with a loan amount above $300,000 or a rate of 6.5% or higher.

```
NumericFieldCriterion amtCri = new NumericFieldCriterion();
amtCri.FieldName = "Loan.LoanAmount";
amtCri.Value = 300000;
amtCri.MatchType = OrdinalFieldMatchType.GreaterThan;
NumericFieldCriterion rateCri = new NumericFieldCriterion();
rateCri.FieldName = "Loan.Rate";
rateCri.Value = 6.5;
rateCri.MatchType = OrdinalFieldMatchType.GreaterThanOrEquals;
StringFieldCriterion stateCri = new StringFieldCriterion();
stateCri.FieldName = "Loan.State";
stateCri.Value = "AZ";
QueryCriterion fullQuery = stateCri.And(amtCri.Or(rateCri));
LoanIdentityList ids = session.Loans.Query(fullQuery);
```

Note that the query above would have very different results if we had rearranged the order of the criteria when combining them with And() and Or(). For example, if we had specified the query as,

```
rateCri.Or(amtCri.And(stateCri))
```

we would receive all loans with a rate of 6.5% or higher or which are both located in Arizona and have a loan amount above $300,000.

Canonical Field Names

Every field that you can query has a canonical name with the format Source.Field. You can use the Source portion when there are multiple sources of data in a single query, as when querying recently completed loans for borrowers or business contacts (see below). The EncompassObjects API supports the following data sources.

- Loan – The source for all fields when performing a query against the Loans database.
- Field – The source for field data stored in the Reporting Database (refer to *Reporting Database Canonical Field Names* for details).
- Contact – The source for all contact-related fields when querying against the Contacts database.
- Opportunity – the source for all fields related to a business opportunity. Business opportunities are tied to Borrower Contacts, so these terms can be combined with fields from the Contact source when querying against the Borrower Contact database.
- Custom – the source used for terms that identify custom fields defined on Borrower and Business Contacts. For example, if a Borrower Custom Field is defined with the label "FavoriteColor", a valid search criterion when querying against BorrowerContacts would be "Custom.FavoriteColor".

The following tables list all of the field names available for use within query criteria.

Loan Object Fields

Field Name	Field Type
Loan.ActionTaken	String
Loan.ActionTakenDate	Date
Loan.Active	String (Y/N)
Loan.Address1	String
Loan.Address2	String
LoanAdverse	String (Y/N)
Loan.Amortization	String
Loan.AppraisalVendor	String
Loan.AppraisedValue	Numeric
Loan.ARMAdjustmentDate	Date
Loan.ARMMargin	Numeric
Loan.ARMLifeCap	Numeric
Loan.ARMFloorRate	Numeric
Loan.ARMFirstRateAdjCap	Numeric
Loan.BorrowerEthnicity	String, comma-delimited
Loan.BorrowerFirstName	String
Loan.BorrowerLastName	String
Loan.BorrowerName	String
Loan.BorrowerRace	String
Loan.BorrowerSex	String
Loan.Broker	String
Loan.BuySideCommited	String (Y/N)
Loan.CensusTract	String
Loan.City	String
Loan.CLTV	Numeric
Loan.CoBorrowerEthnicity	String, comma-delimited
Loan.CoBorrowerFirstName	String
Loan.CoBorrowerLastName	String
Loan.CoBorrowerName	String

Loan Object Fields

Field Name	Field Type
Loan.CoBorrowerRace	String
Loan.CoBorrowerSex	String
Loan.Completed	String (Y/N)
Loan.CreditScore	Numeric
Loan.CreditVendor	String
Loan.CurrentMilestoneDate	Date
Loan.CurrentMilestoneName	String
Loan.DateApprovalReceived	Date
Loan.DateCompleted	Date
Loan.DateCreated	Date
Loan.DateDocsSigned	Date
Loan.DateFileOpened	Date
Loan.DateFunded	Date
Loan.DateOfEstimatedCompletion	Date
Loan.DateOfFinalAction	Date
Loan.DatePurchased	Date
Loan.DateSentToProcessing	Date
Loan.DateShipped	Date
Loan.DateSubmittedToLender	Date
Loan.DocPrepVendor	String
Loan.DownPayment	Numeric
Loan.DTIBottom	Numeric
Loan.DTITop	Numeric
Loan.EscrowVendor	String
Loan.EscrowWaived	String (Y/N)
Loan.FloodVendor	String
Loan.Guid	String
Loan.HazardInsuranceVendor	String
Loan.Investor	String
Loan.InvestorStatus	String
Loan.InvestorStatusDate	Date
Loan.ISStatementDue	Date
Loan.ISPaymentDue	Date
Loan.ISStatementDue	Date
Loan.LastModified	Date

Loan Object Fields

Field Name	Field Type
Loan.Lender	String
Loan.LeinPosition	String
Loan.LOAdditionalCommission	Numeric
Loan.LoanAmount	Numeric
Loan.LoanBrokerProfit	Numeric
Loan.LoanCloserId	String
Loan.LoanCloserName	String
Loan.LoanFolder	String
Loan.LoanName	String
Loan.LoanNumber	String
Loan.LoanOfficerId	String
Loan.LoanOfficerName	String
Loan.LoanOfficerProfit	Numeric
Loan.LoanProcessorId	String
Loan.LoanProcessorName	String
Loan.LoanProcessorProfit	Numeric
Loan.LoanProgram	String
Loan.LoanPurpose	String
Loan.LoanRate	Numeric
Loan.LoanSource	String
Loan.LoanType	String
Loan.LockCommitment	String
Loan.LockDays	Numeric
Loan.LockExpirationDate	Date
Loan.LockRequestDate	Date
Loan.LockRequested	String (Y/N)
Loan.LockRequestPending	String (Y/N)
Loan.LockStatus	String
Loan.LOCommissionByLoan	Numeric
Loan.LOCommissionByProfit	Numeric
Loan.LTV	Numeric
Loan.ManagerProfit	Numeric
Loan.MortgageInsuranceVendor	String
Loan.MSA	String
Loan.NetBuyPrice	Numeric

Loan Object Fields

Field Name	Field Type
Loan.NetProfit	Numeric
Loan.NetSellPrice	Numeric
Loan.NextMilestoneDate	Date
Loan.NextMilestoneName	String
Loan.OccupancyStatus	String
Loan.OtherProfit	Numeric
Loan.PrevMilestoneGroupDate	Date
Loan.ProcessorReviewed	Date
Loan.PropertyType	String
Loan.RateIsLocked	String
Loan.ReferralSource	String
Loan.SecondMortgage	String (Y/N)
Loan.SellSideCommitted	String (Y/N)
Loan.State	String
Loan.Status	String
Loan.Term	Numeric
Loan.TitleVendor	String
Loan.TotalAdditionalCommission	Numeric
Loan.TotalBuyPrice	Numeric
Loan.TotalCommissionByLoan	Numeric
Loan.TotalCommissionByProfit	Numeric
Loan.TotalLoanAmount	Numeric
Loan.TotalSellPrice	Numeric
Loan.TradeGuid	String
Loan.TradeNumber	String
Loan.UnderwriterVendor	String
Loan.Zip	String

BorrowerContact Object Fields

Field Name	Field Type
Contact.Anniversary	Date (recurring)
Contact.Birthdate	Date (recurring)
Contact.BizAddress1	String
Contact.BizAddress2	String
Contact.BizCity	String

BorrowerContact Object Fields

Field Name	Field Type
Contact.BizEmail	String
Contact.BizState	String
Contact.BizWebUrl	String
Contact.BizZip	String
Contact.ContactID	Numeric
Contact.ContactType	String
Contact.CustField1	String
Contact.CustField2	String
Contact.CustField3	String
Contact.CustField4	String
Contact.EmployerName	String
Contact.FaxNumber	String
Contact.FirstName	String
Contact.HomeAddress1	String
Contact.HomeAddress2	String
Contact.HomeCity	String
Contact.HomePhone	String
Contact.HomeState	String
Contact.HomeZip	String
Contact.Income	Numeric
Contact.JobTitle	String
Contact.LastModified	Date
Contact.LastName	String
Contact.Married	Numeric (0/1)
Contact.MobilePhone	String
Contact.NoCall	Numeric (0/1)
Contact.NoFax	Numeric (0/1)
Contact.NoSpam	Numeric (0/1)
Contact.OwnerID	String
Contact.PersonalEmail	String
Contact.PrimaryEmail	String
Contact.PrimaryPhone	String
Contact.Referral	String
Contact.SpouseContactID	Numeric
Contact.SpouseName	String

BorrowerContact Object Fields

Field Name	Field Type
Contact.SSN	String
Contact.Status	String
Contact.WorkPhone	String

Opportunity Object Fields

Field Name	Field Type
Opportunity.Amortization	String
Opportunity.Bankruptcy	Numeric (0/1)
Opportunity.CreditRating	String
Opportunity.DownPayment	Numeric
Opportunity.Employment	String
Opportunity.HousingPayment	Numeric
Opportunity.LoanAmount	Numeric
Opportunity.MortgageBalance	Numeric
Opportunity.MortgageRate	Numeric
Opportunity.NonHousingPayment	Numeric
Opportunity.PropertyAddress	String
Opportunity.PropertyCity	String
Opportunity.PropertyState	String
Opportunity.PropertyType	String
Opportunity.PropertyUse	String
Opportunity.PropertyValue	Numeric
Opportunity.PropertyZip	String
Opportunity.PurchaseDate	Date
Opportunity.Purpose	String
Opportunity.PurposeOther	String
Opportunity.Term	Numeric

BizContact Object Fields

Field Name	Field Type
Contact.BizAddress1	String
Contact.BizAddress2	String
Contact.BizCity	String
Contact.BizEmail	String

BizContact Object Fields

Field Name	Field Type
Contact.BizState	String
Contact.BizWebUrl	String
Contact.BizZip	String
Contact.CategoryID	Numeric
Contact.Comment	String
Contact.Company.Name	String
Contact.ContactID	Numeric
Contact.CustField1	String
Contact.CustField2	String
Contact.CustField3	String
Contact.CustField4	String
Contact.FaxNumber	String
Contact.Fees	Numeric
Contact.FirstName	String
Contact.HomePhone	String
Contact.JobTitle	String
Contact.LastModified	Date
Contact.LastName	String
Contact.LicenseNumber	String
Contact.MobilePhone	String
Contact.NoSpam	Numeric (0/1)
Contact.OwnerID	String
Contact.PersonalEmail	String
Contact.PrimaryEmail	String
Contact.PrimaryPhone	String
Contact.WorkPhone	String

To query for a BizContact based on a business category field use one of the following formats for the field name.

Standard Category:

StandardCategory.<Category Name>.<Field Name>

Custom Category:

CustomCategory.<Custom Category Name>.<Custom Field Name>

The following code uses the custom category “MyCategory” and the custom category field “MyCustomField” (from the “MyCategory” custom category).

```
StringFieldCriterion cri = new StringFieldCriterion();
cri.FieldName = “CustomCategory.MyCategory.MyCustom Field”;
cri.Value = “XYZ”;
ContactList contacts = session.Contacts.Query(cri, Contact LoanMatchType.None, ContactType.Biz);
```

Related Loan Contact Queries

When querying the contact database, you can specify query terms that are compared against one or more of the loans associated with each contact. For information on how to associate a contact to a loan, see the section entitled Linking Contacts to Loans.

When calling the Contacts.Query() method, you can include any of the Loan Object Fields listed above as part of your filter. When you do so, you must also supply an argument which specifies how you want to use these criteria when searching for matching records.

In particular, you must select one of the following options:

- Most recently completed loan only - only the loan that was marked as Completed most recently will be used in evaluating your criteria.
- All completed loans - your criteria must match any single completed loan associated to the contact.
- Most recently originated loan - only the loan created most recently will be used in evaluating your criteria.
- All originated loans - your criteria must match any single loan associated with the contact.

To specify a related loan query, first add one or more criteria that reference the Loan field source.

```
NumericFieldCriterion amtCri = new NumericField-Criterion();
amtCri.FieldName = "Loan.LoanAmount";
amtCri.Value = 300000;
amtCri.MatchType = OrdinalFieldMatchType.GreaterThan;
```

Then specify the related loan match method used in the query at the time you execute it. Provide the correct value of the ContactLoanMatchType enumeration.

```
ContactList contacts = session.Contacts.Query(amtCri, ContactLoanMatchType.LastCompleted, ContactType.Borrower);
```

The above query locates any borrower whose most recently completed loan was for an amount greater than $300,000. You can also combine this with criteria that match against the contact itself, as shown below.

```
DateValueCriterion bdayCri = new DateValueCriterion();
bdayCri.FieldName = "Contact.Birthdate";
bdayCri.Value = DateTime.Parse("1/1/1970");
bdayCri.MatchType = OrdinalFieldMatchType.Less-ThanOrEquals;
bdayCri.Precision = DateFieldMatchPrecision.Year;
ContactList contacts = session.Contacts.Query(amtCri.And(bdayCri), Contact-LoanMatchType.LastCompleted, ContactType.Borrower);
```

The above code locates borrowers born in 1970 or earlier whose most recently completed loan was for more than $300,000.

Contact Custom Field Queries

The EncompassObjects API also permits full query capabilities against borrower and business contact custom fields. These fields provide storage for contact data specific to your company's needs. The ability to locate contacts based on this custom data is an essential part of its value.

Queries against custom fields look exactly like other queries except that they use the Custom data source identifier to specify the FieldName of the query criterion object. The following code uses the custom field “Favorite Color” to locate borrowers whose favorite color is Red.

```
StringFieldCriterion cri = new StringFieldCriterion();
cri.FieldName = "Custom.Favorite Color";
cri.Value = "Red";
ContactList contacts = session.Contacts.Query(cri, Contact-LoanMatchType.None, ContactType.Borrower);
```

You can use custom field criteria by themselves as in the example above, combined with other custom field criteria, or even with non-custom field criteria to form arbitrarily complex queries. For example, the following code locates all borrowers who represent loan opportunities over $200,000 and whose favorite color is Blue.

```
StringFieldCriterion colorCri = new StringField-Criterion();
colorCri.FieldName = "Custom.Favorite Color";
colorCri.Value = "Blue";
NumericFieldCriterion loanCri = new NumericField-Criterion();
loanCri.FieldName = "Opportunity.LoanAmount";
loanCri.Value = 200000;
loanCri.MatchType = OrdinalFieldMatchType. GreaterThan;
ContactList contacts = session.Contacts.Query(colorCri.And(loanCri), Contact-LoanMatchType.None, ContactType.Borrower);
```

Reporting Database Canonical Field Names

When executing queries against the loans in your Encompass system or when extracting loan data as part of a report, you may find the pre-defined list of canonical loan fields does not include one or more fields that are critical to your needs. For example, if your report must include a specific field which is not in the canonical field list, you may have to open each loan to extract the desired field value – a process which can be slow in a system with a large number of loans.

Because there is no way to predict the set of fields that each customer will need for their business processes, Encompass allows you to select the fields you want so you can augment the default canonical field list. By adding a field to the Reporting Database, you will enable your application to use that field as part of a filter (using the QueryCriterion classes) or to extract that field's value for one or more loans without having to explicitly open the associated Loan objects.

Once a field has been added to the Reporting Database, a canonical field name is assigned to that field of the format **Fields.*<FieldID>***. For example, adding the Borrower's Monthly Rent field (field 1005) to the database would allow you use the canonical name *Fields.1005* in any situation in which you would normally use one of the pre-defined canonical loan field values. Similarly, you can create and add custom fields to the database, in which case a field with a name such as CX.MYCUSTOMFIELD would be available with the canonical name of Fields.CX.MYCUSTOMFIELD.

The following example demonstrates how to query loans from the Encompass server by combining a standard canonical field with a field from the Reporting Database. This example assumes you have previously added the field 1005 to the Reporting Database using the Admin Tools application.

```
// First criterion is for a Loan Amount >= $100,000
NumericFieldCriterion laCri = new NumericField-Criterion();
laCri.FieldName = "Loan.LoanAmount";
laCri.Value = 100000;
laCri.MatchType = OrdinalFieldMatchType.Greater-
  ThanOrEquals;

// Second criterion is for a Monthly Rent < $800
NumericFieldCriterion rentCri = new NumericField-
  Criterion();
rentCri.FieldName = "Fields.1005";
rentCri.Value = 800;
rentCri.MatchType = OrdinalFieldMatchType.LessThan;

// Combine into a single criterion and run the query
QueryCriterion queryCri = laCri.And(rentCri);
LoanIdentityList ids = session.Loans.Query(queryCri);
```

Creating Reports

The API provides a variety of different techniques for efficiently extracting loan data from the server. Whether your intent is to generate custom reports or selectively update a set of loan files, choosing the optimal manner by which to access the loan data is critical.

Encompass includes three techniques for quickly searching and retrieving loan data. Each is optimized for a different scenario:

Loans.Query() Method

The Query() method of the EllieMae.Encompass.BusinessObjects.Loans class provides an efficient means of retrieving the identity information for a set of loans that match specific criteria. This method returns a LoanIdentityList object which contains the identifying information for the selected loans. This method is ideal when you need to subsequently perform a specific operation (e.g. assigning a user to a role) on each loan.

Generally, this method is best if you absolutely must open each matching Loan object, for example, to make a change. Although the Query() method is very efficient, the process of opening and modifying multiple Loan objects may be slow.

Advantages:

- Small result set.
- No server resources held.

Disadvantages:

- Often need to open Loan to extract needed data, which is slow.

Loans.QueryPipeline() Method

Like the Query() method above, the QueryPipeline() method allows you to specify a set of criteria to create a filtered set of loans. However, for each loan that meets the criteria, this method returns substantially more information than just the identity information.

In particular, this method returns the data for all of the pre-defined canonical loan object fields as defined in the section *Canonical Field Names*. Reporting Database fields are not included in the result set, although they can be used as part of both the search and sort criteria. The method also returns non-field data which is often needed when rendering a pipeline-type view of the loans, such as information about the current lock held on the loan, if any, and the set of alerts that should be visible to the user.

Because this method could return a considerably larger data set than the Loans.Query() method, the QueryPipeline() function returns its results as a PipelineCursor object. Cursors provide efficient iteration through large result sets by utilizing Encompass Server

resources and by transferring portions of the result set on demand from the server (see Section 9, Pipeline and Contact Cursors for more information on Cursors).

Advantages:

- No need to open Loan if you only need canonical field data.
- Includes non-field data such as the currently held lock and current alerts.

Disadvantages:

- Large data set may include much more data than needed.
- Occupies server resources until closed.

Reporting Objects

The EncompassObjects API provides a set of classes built specifically to help you run efficient reports against the loan database. Those classes, found in the Encompass.EllieMae.Reporting namespace, are accessed through the Session.Reports property.

The most flexible of the reporting methods is Reports.OpenReportCursor(). This method allows you to specify the search criteria and sort order of the resulting data set, as well as the list of the canonical fields that will be included in the results. In this manner, you can control exactly the amount of data returned for the report and minimize data transfer overhead from the server to the client.

Additionally, unlike the methods above, reporting methods allow you to include Reporting Database fields in the result set by using their canonical field names (in addition to the search criteria and sort criteria lists).

The following code example demonstrates how to retrieve the Monthly Rent amount (field 1005) from the Reporting Database for all loans with a loan amount greater than or equal to $100,000. The example assumes you have previously added field 1005 to the Reporting Database through the Admin Tools application.

```
// First criterion is for a Loan Amount >= $100,000
NumericFieldCriterion laCri = new NumericField-Criterion();
laCri.FieldName = "Loan.LoanAmount";
laCri.Value = 100000;
laCri.MatchType = OrdinalFieldMatchType.Greater-ThanOrE-
  quals;

// Build the list of fields to be retrieved. In this case, it's
// field 1005, which comes from the reporting database.
StringList fields = new StringList();
fields.Add("Fields.1005");

// Execute the query to retrieve the report data
LoanReportCursor results = session.Reports.OpenReport-
  Cursor(fields, laCri);
```

Because this method returns a cursor object, it is well-adapted to handling large data sets. However, the cursor does consume a small portion of server memory that is released only when the cursor is closed or the session terminated.

The Reports class also includes other methods that can be used to retrieve field values from a specific loan file or set of loans files (specified by their GUIDs). These methods are far more efficient than explicitly opening the corresponding Loan objects and retrieving the same field values. In general, if you need read-only access to a specific set of fields, using the reporting methods will provide a tremendous performance boost for your EncompassObjects-based application.

Advantages:

- Access to Reporting Database fields.
- Optimized result set contains only the fields you need.

Disadvantages:

- Contains field data only (no lock or alert data).
- Occupies server resources until closed.

Section 7

Calendar and Appointments

The EncompassObjects API provides control of the Calendar feature, including functionality to create, retrieve, modify, and delete appointments. Virtually all calendar-related functionality is exposed through the Calendar and Appointment objects from within the EllieMae.Encompass.BusinessObjects.Calendar namespace. Access to these objects is gained through the Calendar property of the Session object. The Calendar object also provides the ability to search the calendar for appointments within different time intervals.

Retrieving Appointments

The Calendar object, defined in the EllieMae.Encompass.BusinessObjects.Calendar namespace, provides several methods to retrieve appointments from the current user's calendar. The first method retrieves a single appointment based on the appointment's unique ID.

```
Appointment appt = session.Calendar.Open-
  Appointment(175);
```

An Appointment object represents a single appointment on the user's calendar and includes properties such as StartTime and EndTime to designate the times at which the appointment begins and ends. Other properties in the Appointment class include those for setting the appointment's subject and location, and for setting up a reminder for the appointment.

In addition to opening individual appointments, the Calendar object provides methods to retrieve appointments based on other criteria. The first method, GetAllAppointments(), retrieves every appointment, past and present, for the current user. Invoking this method can be fairly expensive if the current user has a long history of appointments. Its use should be avoided unless an entire historical record of the user's appointments is required.

The remaining two functions, GetAppointmentsForDate() and GetAppointmentsInRange(), return the set of appointments that start, end, or are ongoing within a specified time range. The former returns all appointments for a specified date while the latter can be used to specify a custom date range to search. The call,

```
AppointmentList appts = session.Calendar.GetAppointments-
  ForDate(DateTime.Now);
```

can be expressed in its equivalent form,

```
AppointmentList appts = session.Calendar.Get
  AppointmentsInRange(DateTime.Now.Date,
  DateTime.Now.Date.AddDay(1));
```

When retrieving appointments within a specified date range, the API considers the specified range as "closed" with respect to the range's start date and "open" with respect to the range's end date. That is, appointments that start exactly at the time of the range's end date are not considered to be within the range.

Similarly, an Appointment object's EndTime property is considered to be outside of the time span of the actual appointment, while the StartTime is considered part of the appointment's duration. For example, an appointment that ends at 3:00 PM is not be included in a search range that starts at 3:00 PM (unless the appointment also starts at 3:00 PM).

A special class of appointments is "all-day" appointments. These appointments are assumed to start at midnight of the Appointment's specified StartTime and end at the end of the day specified by the appointment's EndTime. Unlike other appointments for which the EndTime is considered outside of the appointments actual range, the EndTime for an all-day appointment actually represents the final full day of the appointment. A single-day, all-day appointment has both its StartTime and EndTime set to the date of the appointment.

Modifying and Saving Appointments

With the exception of the Appointment's ID property, all data stored within an appointment can be modified through the API. Once changes are made, invoke the object's Commit() method to save the appointment. The following code opens, modifies, and saves an existing appointment.

```
Appointment appt = session.Calendar.Open-
  Appointment(175);
appt.Subject = "Meet Joe for Lunch";
appt.Location = "Bayside Diner";
appt.Commit();
```

You can also associate an Appointment object with one or more borrower or business contacts from the contacts database. This association is done through the appointment's Contacts property. Using this property you can add and remove associations between the appointment and any number of Contact objects. For example, the following code associates an existing appointment with a BorrowerContact with a predetermined ID.

```
Appointment appt = session.Calendar.OpenAppo-
  intment(175);
Contact contact = session.Contacts.Open(202);
appt.Contacts.Add(contact);
appt.Commit();
```

Creating and Deleting Appointments

To create a new appointment in the current user's Calendar, simply invoke the CreateAppointment() method.

```
Appointment appt = session.Calendar.Create-
  Appointment(DateTime.Now,
  DateTime.Now.AddHours(1));
appt.Subject = "Urgent Meeting";
appt.Location = "Board Room";
appt.ReminderEnabled = true;
appt.ReminderInterval = 20;
appt.Commit();
```

As with contacts and loans, the new Appointment object returned by the call to CreateAppointment() is not saved to the database until the Commit() method is invoked. The appointment is not created if the commit method is not called.

To delete an appointment, retrieve the corresponding Appointment object and invoke its Delete() method.

```
Appointment appt = session.Calendar.Open-
  Appointment(175);
appt.Delete();
```

Once Delete() is called, the Appointment object becomes invalid and can no longer be accessed. Attempts to invoke properties or methods of the object return an exception.

Section 8

Managing Users

In addition to the ability to manipulate Loan and Contact data within Encompass, the EncompassObjects API provides the ability to perform user management of your Encompass system through your own custom application. The API provides objects and methods for maintaining both your company's organizational hierarchy information as well as the user accounts with which your employees access Encompass.

From the Session object the API exposes two subobjects specific to user management: Session.Users and Session.Organizations. The former can be used to retrieve and modify existing user accounts, delete accounts, enable or disable user logins, and to modify the access rights of your Encompass users. The latter provides for full traversal of the organization hierarchy's tree structure, creation and deletion of organization nodes within your hierarchy, and for the creation of new user accounts.

Managing User Access to Encompass via the SDK

To control which dedicated user accounts in your organization can access Encompass via the SDK, it is recommended that you utilize the Disable Login or Disable Account options in the SDK account's user profile (i.e., the User Details screen in the Organization/Users setting in Encompass Settings). While Encompass provides administrators the ability to force users to change their password when they are logging into Encompass (by selecting the **Force user to change password** option in the user's profile) and they can set an expiration date for a password via the Sever Settings Manager in Encompass Admin Tools, these policies are not enforced for user accounts accessing Encompass via the SDK since there is no physical user in place to actually change the password if prompted.

Managing the Organization Hierarchy

A company's organization hierarchy is logically maintained in a tree structure within Encompass. As such, there is a single top-most (or root) organization which then contains zero or more suborganizations. Each suborganization can, in turn, contain additional suborganizations so that an arbitrarily deep tree can be created.

To traverse the organization hierarchy structure, the first step is to obtain a reference to the top-most node in the tree. The API provides the method Organizations.GetTopMostOrganization() for just this purpose.

```
Organization rootOrg = session.Organizations.GetTopMost-
  Organization();
```

From the root organization, the tree can be traversed by recursively invoking the Organization's GetSuborganizations() method. The method returns a list of the immediate children of the current organization node.

The Organizations object provides other means of retrieving one or more organizations from the organization hierarchy directly without having to navigate to the desired node through the tree structure. The GetOrganization() method provides direct access when the organization's unique ID is known. The GetOrganizationsByName() method allows retrieval based on the organization's name. Note, however, that because Encompass does not enforce uniqueness of names within the hierarchy, this method may return multiple matching organizations. Finally, the GetAllOrganizations() method returns a flattened list of all nodes in the hierarchy.

The Organization object, which represents a single node in the hierarchy, provides the methods necessary for fetching and creating users within the specified branch (refer to "Creating and Deleting Users" below) as well as for creating new suborganizations below the branch.

```
Organization rootOrg = session.Organizations.
  GetTopMostOrganization();
Organization childOrg = rootOrg.CreateSuborgani-
  zation("Loan Officers");
childOrg.Commit();
```

Similarly, you can delete an organization from the hierarchy by invoking the Organization's Delete() method. However, an Organization can only be deleted if it contains no users or suborganizations.

```
OrganizationList orgs = session.Organizations.
  GetOrganizationsByName("Loan Officers");
foreach (Organization org in orgs) org.Delete();
```

Personas

Every user is assigned one or more personas. Personas define the basic rights of the user within the Encompass application, such as whether they can create loans, run reports, etc. In Banker Edition, personas are completely customizable, allowing greater control of individuals' access rights.

The EncompassObjects API does not provide interfaces for creating or modifying personas within Encompass. However, the API does allow you assign personas to both new and existing users. You can gain access to the full set of defined personas through the Session.Users.Personas object.

```
foreach (Persona p in session.Users.Personas)
  Console.WriteLine(p.Name);
```

Every system comes with two built-in personas which cannot be deleted or modified:

- Super Administrator, which provides the user complete control over the Encompass system (equivalent to the "admin" user).
- Administrator, which provides the user local user management within their portion of the organization hierarchy.

Many of the administration functions within the EncompassObjects API require that the currently logged in user have, at a minimum, the Administrator persona (the Super Administrator persona would also satisfy this requirement). For example, creating new users requires the logged in user be an Administrator.

The API provides methods for retrieving all users with a specified Persona by means of the Users.GetUsersWithPersona() method.

```
Persona p = session.Users.Personas.GetPerson-
  aByName("Administrator");
UserList users = session.Users.GetUsersWithPersona(p, false);

foreach (User user in users)
  Console.WriteLine("The user '" + user.ID + "' is an
    Administrator");
```

Note that when you retrieve the users with a given persona you can specify through the second parameter of the GetUsersWithPersona() function whether the user must have only the specified persona or if they can have other personas as well.

Retrieving, Modifying and Saving Users

Retrieving a user's account information can be done in one of several ways. If you know a priori the login ID of the user of interest, you can retrieve the corresponding User object directly:

```
User user = session.Users.GetUser("joe");
```

Because every user is assigned to a specific organization in the organization hierarchy, you may instead want to find all of the users who belong to a specific branch of your organization. Each branch is represented logically by an Organization object within the API, and this object exposes a GetUsers() method to retrieve only those users in that organization.

```
Organization rootOrg = session.Organizations.
  GetTopMostOrganization();
UserList rootUsers = rootOrg.GetUsers();
```

Regardless of how you retrieve an instance of a User object from the API, modifying that object and committing those changes back to the Encompass Server is done in a manner similar to that used for the Loan or Contact objects.

```
User user = session.Users.GetUser("joe");
user.FirstName = "Joe";
user.LastName = "Benson";
user.Email = "joe@mycompany.com";
user.Commit();
```

Any changes you make to the User object are held in memory until you invoke the Commit() method. If you want to discard the changes you have made to the object without re-fetching the object from the server, you may use the Refresh() method.

Because the maintenance of user accounts is an administrator-level function, it is an error for a non-administrator to attempt to modify the properties of or attempt to Commit() a User object except in a few limited cases. In particular, a non-administrative user logged in through the API can update the name, password, email and phone number information for himself and commit that information to the database. Attempts to modify any properties other than these will result in an error.

Creating and Deleting Users

As with modifying users, the ability to create or delete user accounts is reserved for users with the Administrator persona. Because all users must belong to an Organization from the organization hierarchy, creating a user first requires that you obtain a reference to the Organization in which the new user will reside.

Additionally, every user must be assigned to one or more personas. Creating a user therefore requires you to create a list of the Personas to which the user will be assigned.

```
PersonaList personas = new PersonaList();
personas.Add(session.Users.Personas.GetPerson-
  aByName("Sr Manager"));
personas.Add(session.Users.Personas.GetPerson-
  aByName("Administrator"));

Organization org = session.Organizations.GetTop-
  MostOrganization();
User newUser = org.CreateUser("mary", "maryspwd",
  personas);
```

```
newUser.FirstName = "Mary";
newUser.LastName = "Jones";
newUser.Commit();
```

As with Loans and Contacts, the User object returned by the call to CreateUser() is an ephemeral object until you invoke the Commit() method, which saves the new user to the database.

Deleting an existing user requires only that you are able to retrieve the User object for that account.

```
User user = session.Users.GetUser("mary");
user.Delete();
```

Deleting a user is a permanent action and will cause all settings and properties associated with that user to be destroyed. You should delete a user only if you are sure you will never need access to that user's account again.

Managing User Groups (Banker Edition only)

Banker Edition includes the ability to define and assign users to user groups which determine a user's access to a variety of Encompass features. Although the EncompassObjects API does not include the functions required to create or modify the permissions of a user group, it does allow you to assign users to, remove users from or move users between existing groups.

By default every user is assigned to the "All Users" group, a pre-defined group which ships with Encompass. To assign a user to a new group, use the UserGroup.AddUser() method.

```
User mary = session.Users.GetUserByID("mary");
UserGroup group = session.Users.Groups.GetGroup-
  ByName("Managers");
group.AddUser(mary);
```

You can also add an entire element or branch of the organization hierarchy to a user group using the AddOrganization() method. The second argument to this function indicates if the entire organization branch from that point down is added to the group (i.e. a recursive add).

```
Organization topOrg = session.Organizations.GetTopMostOr-
  ganization();
UserGroup group = session.Users.Groups.GetGroup-
  ByName("Managers");
group.AddOrganization(topOrg, false);
```

Similarly, you can remove a user or organization from a UserGroup using the RemoveUser() and RemoveOrganization() methods. However, these methods only remove entities that have been directly added to the group. For example, if a user is a member of a group by virtue of being within an organization that was assigned to the group, calling RemoveUser() for this user will have no effect. You would have to remove that user's organization to remove him from the group.

Similarly, if an organization is included in the group because its parent was added with the recursive flag set to true, you cannot remove it by calling RemoveOrganization().

Managing Loan Officer Licensing Data

For each Loan Officer account, Encompass maintains a list of the states in which the Loan Officer is licensed to originate loans. The EncompassObjects API provides the ability to retrieve and modify the information in this list through the StateLicenses object which is accessible from the User.

The StateLicenses collection maintains a set of StateLicense objects, each of which represents the LO's license to originate in a particular state. You can add to or remove items from this list, or enable/disable an already existing license. The following demonstrates how to add a new license to the user for the state of Washington.

```
User user = session.Users.GetUser("nancy");
StateLicense license = user.StateLicenses.Add("WA");
license.LicenseNumber = "1234ABC78";
license.Enabled = true;
user.Commit();
```

StateLicense objects are indexed using the two-character state abbreviation. And, just as with the other User properties, changes made to the StateLicenses collection are not saved to the Encompass Server until the Commit() method is invoked on the User.

Because a StateLicense can be enabled and disabled through the Enabled property, you can revoke a loan officer's license without having to delete the existing license information from Encompass. For example, to revoke an LO's license to originate in Nevada, you would code the following:

```
User user = session.Users.GetUser("nancy");
StateLicense license = user.StateLicenses["NV"];
if (license != null) license.Enabled = false;
user.Commit();
```

Managing Third Party Originators

The Encompass SDK provides several methods to enable access to the TPO management administrative settings introduced in Encompass 14.2. To view sample code for these methods, please refer to the **EncompassObjects.chm** file located in the *Sdk\Docs* sub-folder in the Encompass SDK installation folder provided on your computer.

For example, here is sample code to demonstrate how the Encompass SDK can be used to change the *Company Legal Name* field located on the Basic Info tab in the Company Information window.

```
using System;
using System.IO;

using EllieMae.Encompass.Client;
using EllieMae.Encompass.Collections;
using
EllieMae.Encompass.BusinessObjects.ExternalOrganization;

class UserManager
{
    public static void Main()
    {
      // Open the session to the remote server.
      // We will need to be logged as an
      // Administrator to modify the user
      // accounts.
      Session session = new Session();
      session.Start("https://be11111.ea.elliemae.net$be11111",
      "admin", "adminpwd");

      //Get external organization based on an external Id
      ExternalOrganization externalOrg =
      session.Organizations.GetExternalOrganization
      ("SampleExternalID");

      externalOrg.CompanyLegalName = "New legal name";
      externalOrg.Commit();

      // End the session to gracefully disconnect from the
      // server
      session.End();
  }
}
```

Section 9

Pipeline and Contact Cursors

Often you may find that your application needs to retrieve a long list of loans or contacts from the server but you cannot afford to pull all of these items to the client because of the performance impact of doing so. To help alleviate this problem the EncompassObjects API includes the Cursor class. A cursor is a static list of items held on the Encompass Server which can be randomly accessed by the API client application. By using cursors you may be able to dramatically improve the performance of your application when handling large data sets.

Scenarios in which you may find using a cursor beneficial include:

- Displaying large lists of contacts or loans that are split across multiple visual pages
- Iterating over large sets of loans or contacts when memory consumption is an issue
- Extracting data for reports based on pipeline data

The API provides two distinct cursor implementations for your use:

- PipelineCursor, which is used to traverse long lists of PipelineData objects
- ContactCursor, which provides fast access to a long list of Borrower or Business Contacts.

Creating a cursor is done by invoking the appropriate API based on the desired cursor type. For example, you can open a PipelineCursor by invoking the Loans.OpenPipeline() method, which returns a cursor containing all of the loans in the user's pipeline.

```
PipelineCursor cursor = session.Loans.OpenPipeline(Pipeline-
  SortOrder.
  LastName);
```

Unlike the Loans.Query() method, which returns a potentially long list of LoanIdentity objects to the client, the list of loans referenced by the PipelineCursor is held in memory on the server. The cursor simply represents a handle which is used to access this list.

Once you have a cursor reference you can retrieve the items from the cursor one at a time using the GetItem() method or in batches using GetItems(). Because each call results in a round trip to the server, you should balance the performance impact of these calls against the memory consumption and interface responsiveness that may be adversely affected by retrieving all of the results at once.

The EncompassObject's cursor implementation also provides an efficient enumerator to allow you to iterate over the contents of a cursor without having to make repeated calls to GetItems().

```
foreach (PipelineData data in cursor)
  Console.WriteLine(data.LoanIdentity.Guid);
```

The enumerator will automatically download the items from the cursor in batches, providing a balance between memory utilization and the expense of the remote server calls.

Because the cursor is a server-side resource, you must be sure to release these resources when you are done with the cursor. This is done by invoking the cursor's Close() method.

```
PipelineCursor cursor = session.Loans.OpenPipeline(Pipeline-
  SortOrder.LastName);
foreach (PipelineData data in cursor)
  Console.WriteLine(data.LoanIdentity.Guid);
cursor.Close();
```

Failure to close a cursor will result in a memory leak on the Encompass Server which will only be cleaned up when the Encompass session is closed.

Using the PipelineCursor

The Encompass pipeline provides a concise view of key fields for all loans accessible to a specific user, such as each borrower's name and loan amount. Because a user's pipeline can contain tens of thousands of loans, the client uses a PipelineCursor to render the pipeline screen quickly and with minimal network traffic.

The EncompassObjects expose these same capabilities to your application. More specifically, you can use any of four methods on the EllieMae.Encompass.BusinessObjects.Loans.Loans class to open a PipelineCursor:

- OpenPipeline(), which opens the user's pipeline using a pre-defined sort order.
- OpenPipelineEx(), which opens the user's pipeline using a custom sort order.
- QueryPipeline(), which allows you to run custom queries against the user's pipeline.
- QueryPipelineEx(), which permits both custom queries and a custom sort order.

Additionally, the LoanFolder object provides its own versions of the OpenPipeline() and QueryPipeline() methods, but the results are limited to the loans in the specified folder.

Once you have opened the cursor, you will want to retrieve one or more of the items in the resulting set. Each data element within the cursor is represented by a PipelineData

object. This object provides high-level information about the loan it references, including the loan's identity (Guid); any alerts currently active on the loan; what, if any, lock is held on the loan; and a list of field values which provide summary data for the loan.

```
PipelineCursor cursor = session.Loans.OpenPipeline(Pipeline-
  SortOrder.
  LastName);
PipelineData data = cursor.GetItem(0);

If (data.CurrentLock != null)
  Console.WriteLine("The loan is currently locked by " +
    data.CurrentLock.LockedBy);
```

Retrieving a PipelineData object from the server is far less resource intensive than retrieving the entire Loan object, so using the PipelineData object when you only need the data it provides is required can save a great amount of time.

Another typical use of the PipelineCursor is in reporting. The PipelineData class provides an indexer which is keyed off of the same field names available to you when you run a query against the loans (see the heading "Canonical Field Names" on page 30). For example, if you need a report listing the Loan Officers for every loan in the pipeline, the following code would do this efficiently:

```
PipelineCursor cursor = session.Loans.OpenPipeline(Pipeline-
  SortOrder.None);
foreach (PipelineData data in cursor)
  Console.WriteLine("LO: " + data["LoanOfficerID"]);
cursor.Close();
```

To help you identify the fields available through the PipelineData class, it provides a GetFieldNames() method that returns a list of all of the fields available to you using the object's indexer.

```
foreach (string fieldname in data.GetFieldNames())
  Console.WriteLine(fieldName + ": " + data[fieldName]);
```

Using the Contact Cursor

Just as a PipelineCursor can be used to efficiently generate or iterate over a large set of PipelineData objects, the ContactCursor can be used to handle large sets of either borrower or business contacts. The Encompass application uses the ContactCursor to efficiently handle the contacts lists on the Contacts tab, only downloading the contacts that are visible on the screen instead of retrieving the full list, which could be prohibitive.

Opening a ContactCursor is done using one of two methods on the EllieMae.Encompass.BusinessObjects.Contacts.Contacts object:

- OpenCursor(), which creates a cursor containing all contacts accessible by the user in a custom sort order.
- QueryCursor(), which can be used to generate a filtered list of the contacts.

Once you have the cursor, the individual element type of the cursor will depend on the ContactType specified in the call that created the cursor. For example, the following code creates a cursor to iterate over a list of borrower contacts.

```
ContactCursor cursor = session.Contacts.OpenCursor(null,
  ContactType.Borrower);
foreach (BorrowerContact contact in cursor)
  Console.WriteLine(contact.FirstName + " " +
    contact.LastName);
cursor.Close();
```

To release the server resources tied to this cursor, call the Close() method.

Section 10

Basic Automation (Banker Edition only)

Automation refers to the ability to write code that directly interacts with or changes the behavior of the user interface. Custom applications written against the EncompassObjects API are not capable of automation because they are run in a completely separate application space from Encompass. In order for code to be automation-enabled, it must run directly within the Encompass application process space, as if it is an actual part of Encompass.

The EncompassAutomation API is the primary framework for authoring automation components. As discussed in Section 1, “Technical Overview” on page 1, there are two types of automation components supported by Encompass: *Custom Input Form Codebase Assemblies* and *Plugin Assemblies*.

Codebase assemblies are written in conjunction with custom input forms built within the Custom Input Form Builder. By using a codebase assembly you can greatly extend the capabilities of your input forms beyond what is permitted using the Form Builder alone. Plugin assemblies are pieces of code that are loaded into the process space when the user logs in and which subscribe to application-level events to perform certain custom actions. Sections describing each integration method can be found later in this document.

Regardless of the technique you employ, the basic automation framework is the same. The objects and their associated properties and methods provide the basis for all code which will interact with the Encompass user interface.

The EncompassApplication Object

When authoring an automation assembly for Encompass you will be referencing the EncompassAutomation assembly within your component. Within this library resides the EllieMae.Encompass.Automation.EncompassApplication class, which acts as the primary entry point for most functions you will want to perform using the automation interface. This class’s properties and methods are all “static,” meaning you can invoke them without having to instantiate the EncompassApplication object explicitly. In effect, the EncompassApplication’s services are available to you anywhere within your assembly to be accessed as needed.

Two of the most basic and useful properties of the EncompassApplication object are the CurrentLoan and CurrentUser properties. The CurrentLoan property can be used to the access the Loan object which is currently open in the Encompass loan editor. The Loan object exposed through this property is the same on provided by the EncompassObjects API, thus all of the capabilities covered in Sections 4 through 6 are available to you in this environment. For example, the following code demonstrates how you might determine if the current loan has been completed or not.

```
LogMilestoneEvents msEvents =
  EncompassApplication.
    CurrentLoan.Log.MilestoneEvents;
MilestoneEvent ms = msEvents.GetEventFor
  Milestone("Completion");
if (ms.Completed)
  // Perform your processing here
```

Similarly, the CurrentUser property provides access to the User object for the currently logged in user. As with the Loan object, the User object is taken directly from the EncompassObjects API, meaning all of the functionality covered in Section 8 is applicable. For example, you can access the currently logged in user’s name as follows:

```
string name = EncompassApplication.
  CurrentUser.FullName;
```

Although the CurrentLoan and CurrentUser provide access to a large part of the EncompassObjects API within your automation component, certain functionality (such as contact management) is not accessible through these interfaces. To provide the full functionality of the EncompassObjects API to you, the EncompassApplication object provides a Session property, which provides direct access to the underlying client session. Through the Session, you can access every feature of the EncompassObjects API, including the calendar, contacts, and queries.

Another useful property of the EncompassApplication is the CommandLineArguments property, which allows your plugin or codebase assembly to directly inspect the parameters passed to the Encompass.exe application on the command line. You can use this property to launch Encompass with parameters that are meaningful to your automation objects and can affect their behaviors. For example, you could pass a parameter to indicate that Encompass should launch directly into a specific loan file. You can then write a plugin assembly that reads this parameter and uses the automation interface to open the specified loan.

Application Events

The EncompassApplication also provides several events which are most useful for plugin assemblies that need to perform specific actions when the user activates a certain

portion of the user interface. For example, your plugin may start a synchronization process whenever the user closes a loan file within the loan editor.

The EncompassApplication provides the following events:

Login	Occurs when the user is fully logged in an the interface is ready for user interaction.
Logout	Occurs after the user has logged out of the server or has been otherwise disconnected from the server. When this event occurs the server connection is already closed so no actions should be taken that require data be pulled from or saved to the server.
LoanOpened	Occurs when the user has opened a loan in the Encompass loan editor.
LoanClosing	Occurs when the user has elected to close the loan currently loaded into the load editor. This event occurs before the loan is actually closed but after the user has saved any changes.

Besides the application-level events, many of the objects within the EncompassObjects API provide their own events which can also be used within your automation components. For example, you can use the Loan object's FieldChange event to detect changes to individual fields in the open loan. For an example of using the EncompassApplication's LoanOpened and LoanClosing events in conjunction with the Loan object's FieldChange event, see the LoanMonitorPlugin sample project included with the SDK.

Manipulating the User Interface

Up to now the discussion has focused on the manipulation of data objects from within the automation framework, such as accessing the current loan or currently logged in user's data. The EncompassAutomation API also provides methods for directly manipulating the user interface.

The majority of this functionality is accessed through the EllieMae.Encompass.Automation.Screen object. Each major screen in Encompass is represented by an instance of this class or a derived class. Access to the individual screen objects is obtained through the EncompassApplication object's Screens property. The following code demonstrates how to obtain a reference to the Borrower Contacts screen and make that screen the currently active one within Encompass.

```
ApplicationScreen s =
  EncompassApplication.Screens[Encompass
    Screen.BorrowerContacts];
s.MakeCurrent();
```

Several of the Encompass screens support screen-specific properties or methods, such as the ability to instruct the Loans screen to open a specific loan in the loan editor. To access these functions you must first cast the Screen object to the appropriate derived screen type.

```
LoansScreen s = (LoansScreen)
  EncompasApplication.Screens[Encompass
    Screen.Loans];
s.Open("{a2f3d228-8bed-4e39-9eaa-f5f07db3c272}");
```

The "EncompassObjects API Reference" document (help topic) that accompanies the SDK includes the definitions of the properties and methods supported by each derived Screen class. These methods include:

- The ability to open a loan in the loan editor or to close the loan editor
- The ability to navigate to a particular URL in the Ellie Mae Network window
- The ability to filter or modify the current view of the Pipeline screen
- The ability to filter the list of contacts displayed on the Borrower or Business contacts screens

NOTE: *You can access the "EncompassObjects API Reference" document from the Encompass SDK program group in the Start menu in the SDK.*

Macros

The EncompassAutomation API includes a set of functions, called macros, which can perform common actions quickly with a greatly reduced amount of coding. Although macros are targeted at users who are writing code within the Form Builder's event editor, any code written against the API can take advantage of these functions.

All of the macros can be found as static methods on the EllieMae.Encompass.Automation.Macro class. Thus, invoking a macro anywhere in your code is very easy. For example, the following code to copy the value of one loan field to another within the loan currently open in the editor takes only one line.

```
Macro.CopyField("1109", "CX.MYLOANAMT");
```

The list of all available macros is provided in the *Encompass Input Form Builder User's Guide* (Chapter 10). However, a few of the macros which may be particularly useful from within your automation assemblies are noted below.:

Eval(string expression)	Evaluates an expression using the Custom Calculation Engine against the currently open loan file.
GoToScreen(EncompassScreen screen)	A short-hand method for setting the currently visible screen in the user interface.
OpenURL(string url)	Opens the specified URL in a pop-up browser window using the user's default web browser.
SendKeys(string keys)	A method for sending keystrokes to the UI to take advantage of behaviors not exposed through the API.

Section 11

Plugins (Banker Edition only)

A Plugin is an application extension that runs directly within the client and monitors for specific events to occur either from within the application or from an external source. A plugin may perform synchronization or integration tasks without the knowledge of the end user, or it may provide a custom user interface which provides functionality beyond what is built into Encompass.

Regardless of its intended behavior, every plugin is created and loaded in the same manner. Section 3, “Getting Started” on page 6, discusses the basic steps for generating a custom plugin class and installing into your Encompass system. What follows is a brief architectural overview of the plugin system as well as recommendations and best practices for developing reliable plugins.

Plugin Architecture

When a user successfully logs into Encompass, one of the tasks performed by the client application is to query the Encompass Server for any available plugins. The server provides file name and version information for all DLL files within the EncompassData\Data\Plugins folder configured on the server. The client maintains a local cache of plugin DLLs against which it compares the version information and, if the server’s version is greater than that of the cached version (or no cached version exists), the client downloads the assembly from the server and stores it in the local cache.

Of note during this process are two issues. First, the version information used for determining if the assembly has been updated is the *file version* and not the *assembly version*. By default, these values will be the same but it is possible to provide distinct file and assembly versions by applying both the AssemblyFileVersion and AssemblyVersion attributes to your .NET assembly. This may be useful if you need to maintain the AssemblyVersion number for backwards compatibility with other projects but wish to update the AssemblyFileVersion to identify a new build.

The second note is that the client segregates its local plugin cache using the unique system identifier stored within your database. Thus, if the client connects to a different Encompass system there is no potential for crossover of plugins between them. As a result, the plugins loaded by Encompass are completely determined by the Encompass system to which the client logs in.

Once the client has pulled the plugin assemblies to its local cache, it loads each into memory using the Assembly.Load() method. The assembly’s contents are inspected and, for each class which is decorated with the EllieMae.Encompass.ComponentModel.PluginAttribute attribute, a single instance is constructed by invoking the class’s default, parameterless constructor via reflection. It is a requirement that every plugin class provide such a constructor; otherwise, Encompass will fail to create the plugin instance and will simply skip it.

Another important aspect of the default constructor is that it should register for one or more events from the EncompassApplication class or some other source. After Encompass instantiates the plugin class it immediately releases its reference to it. If you have not registered your class for at least one event or used some other technique to ensure that a root reference is held to your plugin’s instance, your plugin class will be eligible for garbage collection and will be destroyed.

Within your constructor you should never attempt to perform any other operation against the APIs other than registering for events. At the time the plugin is instantiated the main user interface has not yet been loaded. If your plugin needs to be notified at the earliest available moment after the interface is loaded, it should subscribe to the EncompassApplication.Login event. During the processing of this event it can be assumed to be safe to access the full EncompassAutomation object model.

Using Encompass Data Exchange

Many common uses of Encompass plugins require inter-process communication (IPC) between an external system, such as a call center application, and the client. Using standard IPC techniques such as .NET remoting can be difficult or impossible if the application triggering the events doesn’t have knowledge of or access to the host on which the client application is running. For example, consider a call center application that needed to pop up a message on an Encompass user’s interface when a call is received. The call center application has no knowledge of the computer on which the recipient may be logged in at any given time (or know if they even are logged in currently).

The Encompass Data Exchange facilities provide a simple mechanism to leverage the API to perform efficient data exchange between two client systems, one client and one external system or even two external systems using the EncompassObjects API. The design of the system is quite simple: the EncompassObjects’ Session object provides three methods for initiating a data exchange: PostDataToUser, PostDataToSession and PostDataToAll. Each method provides a way to pass an arbitrary data object from the source system to one or more recipients logged into the same Encompass server.

For example, a call center application which integrates the EncompassObjects API may package up all of the caller's information into a delimited string and then invoke PostDataToUser() to send the data to a specific user.

```
string data = "John,Homeowner,555-555-5555";
session.PostDataToUser("mary", data);
```

If you have written a plugin which subscribes to the Session object's DataExchange event you can receive that event within Encompass and display a pop-up message with the caller's information. The plugin could even create a new loan file, populate it with the caller's data and open it in the loan editor.

```
[Plugin]
public class MyPlugin
{
  public MyPlugin()
  {
    EncompassApplication.Session.DataExchange +=
      new DataExchangeEventHandler(onData
        Exchange);
  }

  private void onDataExchange(object sender, Data
    ExchangeEventArgs e)
  {
    MessageBox.Show("Data received: "+ e.Data);
  }
}
```

All data exchange invocations occur asynchronously, meaning that when you one of the Post methods listed above, the call will return before the recipient(s) may have processed the data exchange. As a result, there is no way for the caller to receive a response from the recipients in a synchronous manner. The Post methods do, however, return an integer value to the caller indicating the number of client sessions to which the data was posted. Whether or not any or all of these sessions performed any action (or was even configured to receive) the post cannot be inferred by this value.

If your application requires this form of two-way communication, the recipient will need to post its own data exchange response back to the originating session. This can be done easily since the recipient knows the unique identifier of the session from which the data was posted. It is then up to you to manage the bidirectional communication between the two sessions.

The SDK includes a sample project which uses the Data Exchange interface to simulate a very simple data transfer from a call center. This VS.NET solution can be found in the SDK\Samples\C#\DataExchangeExample folder within your primary installation folder. The example consists of two parts: a sample application which posts data using the Data Exchange interfaces and a sample plugin which receives those posts and creates new loans in the Encompass UI.

Threading Considerations

Because a plugin can be used to process ansychronous events, whether using the Data Exchange facilities provided by Encompass, the .NET remoting functionality or some other technique, your plugin's code may at times run on a thread other than the primary UI thread of the Encompass application. In any Windows application is it essential that, for thread safety, you only access or modify the properties or methods of an UI element from the application's UI thread.

To assist you in ensuring method calls are running on the proper thread, the EncompassApplication.Screens object implements the ISynchronizeInvoke interface, which exposes the InvokeRequired property as well as the Invoke, BeginInvoke and EndInvoke methods. These methods can be used to marshal any function call to the UI thread in either a synchronous (using Invoke) or asynchronous (using BeginInvoke and EndInvoke) manner.

In general, if your plugin subscribes only to events from the EncompassApplication class, it will not have to worry about threading issues since these events are guaranteed to be called on the UI thread. However, if you subscribe to events from classes provided by the EncompassObjects library (such as the Session class), then you will need to write you code in a manner that considers thread safety when accessing automation objects.

For example, consider the following plugin code which catches DataExchange events from the underlying Session and uses them to update the loan currently open in the loan editor.

```
[Plugin]
public class PluginClass
{
  public PluginClass()
  {
    EncompassApplication.Session.DataExchange +=
      new DataExchangeEventHandler(onData
        Exchange);
  }

  private void onDataExchange(object sender,
    DataExchangeEventArgs e)
  {
    EncompassApplication.CurrentLoan.Fields
      ["1109"].Value = e.Data;
  }
}
```

The example above is not thread-safe. The DataExchange event will occur on a thread other than the UI thread, meaning the EncompassApplication.
CurrentLoan object was created in and is read and written to by the loan editor on a different thread. The user could be in the process of modifying the loan (or even closing or saving the loan) when this event occurs.

To make this code thread safe, use the Invoke() method.

```
private void onDataExchange(object sender, Data
  ExchangeEventArgs e)
{
  if (EncompassApplication.Screens.InvokeRequired)
    EncompassApplication.Screens.Invoke(new
      DataExchangeEventHandler(onDataExchange),
      new object[] { sender, e });
  else
    EncompassApplication.CurrentLoan.Fields
      ["1109"].Value = e.Data;
}
```

In the revised code a check is performed to ensure that the event handler is running on the UI thread. If not, the call is marshaled to that thread using the Invoke() method.

External Dependencies

If your plugin depends on one or more external assemblies (other than the Encompass assemblies), you will need to ensure that those assemblies are available to the plugin at run time on the client. To do so, you must place these assemblies in the same Plugins folder on the Encompass Server as your actual plugin assembly. When the client connects to the server it will load all of the assemblies in this folder regardless of whether they contain an actual plugin or not.

However, because the plugin assemblies are loaded in a non-deterministic order, you must not include any code which references external dependencies within your constructor's plugin. At the time the plugin class is instantiated you cannot be guaranteed that all of the assemblies from the Plugin folder have been loaded into memory. If you need to perform some initialization with the external DLLs, you will need to subscribe to the EncompassApplication.Login event and perform the initialization at that point. When the Login event fires you are guaranteed that all of the assemblies from the Plugins folder have been loaded in the process space.

Section 12

Custom Input Form Codebase Assemblies (Banker Edition only)

Using the Input Form Builder included with the Banker Edition, you can create custom input forms which provide unique *layouts* of your users' loan data based to match your company's business needs. The Form Builder's built-in event editor allows for further customization of the *behavior* of the form so you can provide your own simple pieces of functionality based on user events, such as the click of a button or the modification of a field value.

From a development point of view, the ability to author code to create customized behaviors within the Form Builder is extremely limited. To satisfy the need to create more powerful and functional custom forms, Encompass includes the ability to associate a Custom Codebase Assembly with your input form. Within such an assembly you can utilize the full power of .NET 2.0 to provide highly dynamic and customizable interfaces which can interact both with the Encompass application as well as external data sources or resources.

Developers familiar with the ASP.NET Form-based programming model will see very a very strong parallel in the architecture of a custom form codebase assembly. A codebase assembly must always contain at least one class that derives from the EllieMae.Encompass.Forms.Form class. By extending this class, which is the base class for all Encompass input forms, you can intercept the events that are triggered when the form loads, as data is moved between the form and the underlying loan file or when the user performs specific actions such as clicking a button.

Just as in ASP.NET, every input form can be thought of as a set of nested Control objects, the definitions for which are found in the EllieMae.Encompass.Forms namespace within the EncompassAutomation assembly. Examples of controls found on virtually every form are the TextBox, Label and Button controls. Each control type has a distinct set of properties and methods that you can manipulate both at design time, from within the Input Form Builder, and at runtime, within your codebase assembly. For example, you can hide or show a TextBox control (or any other control) simply by setting its Visible property.

Besides manipulating the properties of the form's control elements, a custom codebase assembly can directly access and modify the data from the underlying loan file. Classes which derive from the base Form class can access the base class's Loan property, which represents the currently active loan file. This Loan object is the same as the one defined in the EncompassObjects, thus all of the information covered in Sections 4 through 6 apply equally to this interface.

The third basic capability of the codebase assembly is to interact with other areas of the Encompass application, such as the Contacts screens or Ellie Mae Network. By accessing the static properties and methods of the EllieMae.Encompass.Automation.EncompassApplication class, your codebase assembly can perform a variety of tasks such as:

- Changing the currently displayed screen, for example to display the borrower contacts.
- Perform queries to populate the contacts lists with a specific contact or set of contacts.
- Launch the Ellie Mae Network Services menu for a specific service provider type, such as credit reporting services.

For an introduction on how to get started creating a Custom Form Codebase Assembly and how to associate it with your custom input form, see "Creating a Custom Form Codebase Assembly (Banker Edition only)" on page 10.

Forms & Controls

Every input form, whether it is a standard form such as the 1003 or a custom form created in the Input Form Builder, is represented at runtime by an instance of the EllieMae.Encompass.Forms.Form class, which is part of the EncompassAutomation API. The base Form class provides the basic mechanisms for handling user input and binding data from the loan file to the individual controls.

By creating a codebase class for your custom input form, you are providing Encompass with a derived Form type which will be instantiated whenever the form is loaded into the Encompass loan editor. Each time the form loads, a new instance of your class is constructed so instance data is not preserved between form activations.

The most trivial codebase class would include the following minimum code:

```
public class MyForm : EllieMae.Encompass.Forms.Form
{
}
```

This empty class definition is sufficient to act as the codebase for a form created in the Form Builder, although it provides no new functionality. In order to add functionality, you need to access and manipulate the individual elements for the form.

Each Form instance maintains a hierarchy of controls just as you would find in a Windows Forms or ASP.NET form. You define the initial control hierarchy when you create your form within the Form Builder. For example, the top-level Form control may contain a Panel control. That panel may contain one or more TextBox controls and perhaps some Labels. The Panel may also include within it another Panel, and so on. Certain controls, such as the Panel, CategoryBox and GroupBox, can contain other controls within them to create a tree-like hierarchy of nested controls. Controls which can contain other controls are called Container Controls. The Form object is a container control and is always the top-most element in the control tree.

Every control on an input form has a unique Control ID assigned to it. The Form Builder assigns each new control a default Control ID when it is first added to the form. Using the Properties panel you can change a control's ID to something more descriptive. For example, if you had a TextBox which was meant to show the loan amount, you may give that control the ID *txtLoanAmount*.

Within your derived codebase class you can access control elements on the form using their Control IDs and the inherited FindControl() method. For example, the following code locates the TextBox described above and hides it from the user's view.

```
TextBox t = (TextBox) FindControl("txtLoanAmount");
t.Visible = false;
```

Note that the return value of FindControl() is cast to the class appropriate for the particular control type. Each control type is represented by it own class within the EllieMae.Encompass.Forms namespace, but all control types derive from a common base class: EllieMae.Encompass.Forms.Control. Note that even the Form object itself derives indirectly from this base Control class.

Besides manipulating the properties of the controls to cause particular visual effects, you may also need to subscribe to particular control events. For example, if you have placed a button on the form with the control ID of *btnQuote*, you can provide custom code to handle its Click event as follows.

```
Button b = (Button) FindControl("btnQuote");
b.Click += new EventHandler(btnQuote_Click);
```

When the user clicks the button in the form, your codebase class's *btnQuote_Click* method will be invoked.

For a complete reference of the properties and events available on the individual controls, see the *EncompassObjects API Reference* document which is accessible from the Encompass SDK program group in the Start menu after you install the SDK.

Form Lifetime

When your custom input form is loaded inside the application, Encompass detects that you have associated a custom codebase class and will automatically download it from the Encompass Server. Once downloaded, the form passes through the following stages:

Stage 1: Instantiation

Each time your form is loaded Encompass instantiates a new instance of the codebase class. Your class must include a public, parameterless constructor, which will be invoked by Encompass. Your constructor should not attempt to access the object hierarchy of the form or invoke any other form properties or methods since, at the time of construction, the control tree has not yet been created. You can perform internal initialization of any kind which is needed within your constructor.

Stage 2: Control Generation

Once the class is instantiated, Encompass builds the control tree from the design created in the Form Builder. Upon completion, Encompass invokes the Form's CreateControls() method, which you may override to perform functions such as:

- Attach event handlers to control events
- Create dynamic controls
- Store references to controls in instance variables for quick access elsewhere in your class definition.

At the time of control generation the form has been attached to its underlying Loan file so it is possible to use the Form class's Loan property to access the data from the underlying loan. However, data binding has not yet occurred (see below), meaning that the controls' values have not been set to represent that data in the underlying loan file.

Stage 3: Data Binding

Data Binding is the process by which Encompass populates the *field controls* on the form with the appropriate value from the underlying loan file. A field control is any control type which derives from the EllieMae.Encompass.Forms.FieldControl class. These controls are used to allow user manipulation of data within the loan file. Examples of field controls are TextBox, CheckBox, DropdownBox and RadioButton controls.

When the form is first instantiated, these controls are created in their default state. For example, TextBox controls do not contain any text. For each field control, Encompass invokes the control's BindTo() method to load the data from the loan into the control. Each field control's Field property defines the data field from the loan which is mapped into the control's display. For example, a TextBox that has its Field property set to field 1109 (loan amount) will be loaded with the formatted representation of the loan amount, for example, "230,000.00".

Stage 4: Load Event

Once data binding is complete, Encompass fires the Load event on the Form. If you have subscribed to the Load event from within the class's constructor or CreateControls() method, you can perform additional custom processing at this point. Because data binding has been completed you can safely retrieve or update the contents of the controls on the form. This is also the first point at which it is safe to manipulate the Encompass user interface using the EncompassApplication object.

Stage 5: User Interaction

Once the form has completed loading, it is now ready for the user to navigate its fields, modify the field controls' contents, click buttons, and so on. During this phase your only option for altering the form's default behavior is to subscribe to one or more control-specific events. For example, you can subscribe to a Button control's Click event or a TextBox control's Change event.

This phase of the form's lifetime is typically the longest and the one in which your codebase class will perform the bulk of its work.

Stage 6: Unload Event

Eventually, the user will leave the input form by navigating to another form, closing the loan editor or even logging out of Encompass. When any of these events occurs your form will have one last opportunity to perform custom processing by means of the Unload event. This event, which is part of the base Form class, is fired by Encompass just before the form is unloaded from the editor. You cannot stop the form from being unloaded through your code so this is not an appropriate location to perform validation that must be attended to by the user.

Once the form is unloaded, Encompass will no longer hold a reference to the form's instance, making the object subject to garbage collection unless you have performed some action to prevent this. In general, you should not subscribe to application-level events, i.e. those exposed by the EncompassApplication object, within your form class unless you are sure to unsubscribe to those event within your form's Unload event handler. Otherwise, your form will never be garbage collected and can accumulate in memory as the user moves between forms.

Dynamic Control Creation

When you create a custom input form in the Form Builder, you are creating a static layout which generally does not change at run time. Using a codebase assembly you can dynamically modify your form at run time based on data within the loan, data received from an external source, and so on.

To add a new control to the form, you create a new instance of the desired control type and then add it into a ContainerControl's Controls collection. For example, the following code creates a new TextBox and adds it to a Panel control named *pnlBorrower*.

```
TextBox t = new TextBox();
t.ControlID = "txtDynamicTextbox";
Panel pnlBorrower = (Panel) FindControl
  ("pnlBorrower");
pnlBorrower.Controls.Insert(t);
```

Next, you will likely need to position the control within the form. You can do this by setting either the control's Position or AbsolutePosition properties. The former sets the control's position relative to the top-left corner of the *client area* of the control's container. In this case, the control's container is the pnlBorrower control so the code

```
t.Position = new Point(5, 10);
```

would place the control 5 pixels from the left and 10 pixels from the top of the panel's top-left corner.

The AbsolutePosition places the control in a position relative to the top-left corner of the entire form, even if this location is outside of the bounds of the control's container. For example, the code

```
t.AbsolutePosition = new Point(20, 150);
```

would place the TextBox 20 pixels from the left and 150 pixels from the top of the entire form. Depending on the location of the pnlBorrower panel, the TextBox may actually appear outside of this control's client area. Despite this fact, the control is still contained within the Panel.

Because a TextBox control is a field control, you will most likely want to assign its Field property to map the control to a field within the loan file. The following code maps the control to the loan amount field, field 1109.

```
t.Field = EncompassApplication.Session.Loans.Field
  Descriptors["1109"];
```

If you have created this control within the CreateControls() method of your form, the control will be automatically populated with the field's value during the Data Binding phase of the form's lifetime. However, if you add this control after the Data Binding has occurred (for example, in the Load Event), then you need to force the control to be bound to the loan field by invoking the control's Refresh() method.

```
t.Refresh();
```

Without calling Refresh the control will appear empty on the screen. Refreshing the control forces it contents to be re-synchronized with the underlying loan. All field controls support the Refresh() method.

Other control types have additional property values that you will generally want to set when you create a dynamic control. If your control were a Label or Button, you would need to set the control's Text property to set the label on the control.

```
Button b = new Button();
b.ControlID = "btnDynamic";
Form.Controls.Insert(b, new Point(20, 30));
b.Text = "Click Here";
b.Click += new EventHandler(btnDynamic_Click);
```

Note that in all of the above examples the properties of the control, with the exception of the Control ID, are all set after the control is added to the form. This is a requirement of the EncompassAutomation architecture and attempts to set properties prior to adding the control to the form will result in exceptions.

Event Handling

Many of the control types provided by the EncompassAutomation library support events to allow your codebase class to intercede in the normal behavior of the form. The following table lists the various events that can be caught and how they are used.

Event	Supported By	Description
FocusIn, FocusOut	Field Controls	Fired when the cursor focus enters or leaves a field.
Change	TextBox, DropdownBox, DropdownEditBox	Fired when the user modified the value of the field and then leaves the field. This event is fired before the FocusOut event.
Click	CheckBox, RadioButton, Button, FieldLock, Hyperlink, ImageButton	Fired when the user clicks on the button or control.
Load, Unload	Form	Fired when the form has been load or is being unloaded.
DataBind, DataCommit	Field Controls	Fired when Encompass performs data binding on the control. See the heading “Custom Data Binding” below.
Format	TextBox, DropdownEditBox	Fired as user types into a control, allowing for dynamic formatting of user input.
Popup	PickList	Used to dynamically populate the contents of a PickList.
ItemSelected	PickList	Fired when the user selects an item from a PickList.

As discussed in the sections above, the CreateControls() method is an ideal location to hook up your event handlers for the form’s controls. Keep in mind that when you handle an event you are not limited to using the functionality provided by the EncompassAutomation framework. Your custom event handler could invoke a web service, display a Windows Form dialog, launch a web page or run an external application.

For more information about the different control events provided in Encompass, see the *Encompass Input Form Builder User’s Guide*.

Custom Data Binding

Data binding is the process by which a field control on the form is tied to a field in the underlying loan file so that the value displayed in the control represents what is stored in the loan and changes to the control are saved back into the loan. Data binding is generally handled automatically by Encompass. When the form first loads it binds each field control to the loan field defined by its Field property. Similarly, when the user modified the value in the control, Encompass automatically pushes that change into the corresponding loan field (and triggers any required calculations).

There may be times, however, when you would like to modifies the default behavior of the Encompass data binding and implement your own logic. For example, take the case of field 981, which can be found in the Declarations section of the 1003 Page 3 form (line m(1)). The possible values for this field are: "PrimaryResidence", "SecondaryResidence" and "Investment". However, the control as displayed on the form accepts the values "PR", "SH" and "IP". In this case, the form is overriding the default data binding for this control to translate the field value PrimaryResidence to the control value "PR". If the default data binding were used, the TextBox control would contain the entire value "PrimaryResidence".

Overriding a control’s default data binding behavior consists of handling two events: DataBind and DataCommit. The DataBind event is fired any time the Encompass UI prepares to populate the control with the value from the underlying field. In the case of field 981, the form provides the following event handler for this field:

```
private void onDataBind(object sender,
  DataBindEventArgs e)
{
  if (e.Value = "PrimaryResidence")
    e.Value = "PR";
  else if (e.Value = "SecondaryResidence")
    e.Value = "SH";
  else if (e.Value = "Investment")
    e.Value = "IP";
  else
    e.Value = "";
}
```

The event handler simply provides a translation of the default value obtained from the loan into a new value which is then bound to the control.

Of course, this code only handles one direction of the translation. The reverse process is handled within the event handler for DataCommit. The DataCommit event is fired any time Encompass needs to move data from the control back into the loan. The DataCommit event handler for this controls is implemented as follows:

```
private void onDataCommit(object sender,
  DataCommitEventArgs e)
{
  if (e.Value = "PR")
    e.Value = "PrimaryResidence";
  else if (e.Value = "SH")
    e.Value = "SecondaryResidence";
  else if (e.Value = "IP")
    e.Value = "Investment";
  else
    EventArgs.Value = "";
}
```

In this example, the desired behavior was a simple translation of values. By modifying the Value property of the event arguments, Encompass bound the control to a different value than the one in the underlying loan.

There may be times when you need to short-circuit the data binding process altogether. For example, you may wish to prevent a value from being committed to the loan altogether if it doesn't meet a custom requirement coded into your assembly. In that case, you can set the event argument's Cancel property to true. Doing this will cause Encompass to abort the DataBind or DataCommit processing altogether.

Conversely, there may be times when you need to force the DataBind event to be fired. For example, if you directly modify the value of a loan field using the Form.Loan.Fields property, any control associated with that field will need to be re-bound in order to reflect the updated value. You can force data binding for a particular control by calling its Refresh() method. Alternatively, you can call the Form's Refresh() method, which will refresh all controls for which the underlying field value has been modified since last bound to the control.

Packaging and Redistribution

When developing a codebase assembly for a custom form, you will most likely want to do the initial form creation and assembly debugging on an Encompass system other than your production system.

Once you have tested your form and accompanying codebase assembly, you will need to deploy both items to your production Encompass system. The Encompass Form Builder provides tools specifically to facilitate this deployment.

From the Form Builder's Tools menu you will find the Package Publishing Wizard. This wizard-type interface walks you through the process of posting one or more custom input forms, along with their associated codebase assemblies and custom field definitions, from one Encompass system to another. The Publishing Wizard assumes that you have direct login access to both the source and target Encompass systems.

If you do not have login access to the target system, or if you are a licensed Encompass API Partner who wishes to distribute your custom form to multiple customers, you can instead use the Package Export and Import Wizards. The Export Wizard allows you to generate a single-file export package which contains any number of custom forms, codebase assemblies and custom field definitions. Conversely, the Import Wizard can be used to import these items from an exported package into an Encompass system.

For more information about using these packaging features to deploy custom input forms, see the *Encompass Input Form Builder User's Guide*.

Section 13

Error Handling in the API

A variety of potential error sources can occur when using the EncompassObjects API, and you should take precautions to ensure your application handles them properly. Exceptions raised by the API generally fall into two categories: rules violations and systemic errors.

Rules Violations

Rules violations occur in cases such as when you attempt to open a loan for which the currently logged in user does not have sufficient access. These types of exceptions are often avoidable if you ensure that your code does not perform actions that can potentially generate such errors. Rules violations are not fatal errors and you can write appropriate recovery code to handle these types of errors.

Systemic Errors

Systemic errors are more unpredictable and can arise from problems such as an unexpected disconnection between the client application and the Encompass Server. Any operation that communicates with the server has the potential to result in such an exception. To help detect such errors, the API provides a Disconnect event on the Session object which is raised when a communication error occurs. You can also handle such exceptions locally in the calling function or subroutine, in which case you should set up the error handling mechanism appropriate to your programming language.

Both rule violations and system errors occur as exceptions in your application that can be caught and examined to determine their source and cause. Some of the rules violation errors can be detected by examining the type of exception raised. For example, a LockException is thrown when attempting to lock a loan which is already locked by another user. Such an exception could be handled as follows:

```
try
{
  loan.Lock();
}
catch (LockException ex)
{
  MessageBox.Show("The loan is already locked by " +
    ex.CurrentLockHolder);
}
```

Section 14

Security in the API

Because it is the responsibility of the Encompass Server to ensure the security of the Encompass data held by it, all security rules enforced by the server apply to client applications written using the EncompassObjects API. The first layer of security enforced by the server is that all clients must have a valid user ID and password. Thus, the first step of virtually any client application is to log in to the server (via the Session object) using either hard-coded credentials or credentials obtained at run time.

When a user logs in to the Encompass Server, the server uses the log in credentials and the permissions set for the user to determine if an action is permitted and which sets of data the user can access. For example, the following code opens a connection and retrieves the contents of the “My Pipeline” loan folder.

```
Session s = new Session();
s.Start("myserver", "mary", "maryspwd");
LoanIdentityList ids = s.Loans.Folders["My Pipeline"].GetContents();
```

Because the user “mary” is logged in, the set of loan identities returned by the GetContents() call includes only those to which “mary” has access. If a different user logged in, the server would likely return a different set of loan identities.

Certain operations are only accessible to users with sufficient privileges. For example, the Encompass Server only permits a user with administrative privileges to delete a loan. The following code results in an exception if the logged in user does not have the required permissions.

```
Loan loan = session.Loans.Open(myLoanGuid);
loan.Delete();
```

In general, your application will safely perform every operation provided only if the logged in user has administrative privileges. You should use such a login when writing client applications that run as services to perform data integrations or reporting. Conversely, you should log in as a non-administrative user to ensure that your client application cannot perform unauthorized actions against the Encompass Server.

Section 15

Distributing Applications Built Using the API

Encompass API Run-Time Licensing

The Encompass API run time is a licensed product of Ellie Mae granted either in conjunction with your Encompass license or as part of an Encompass SDK Partner Agreement. The Encompass Software SDK License Agreement (provided in the installation wizard when installing the SDK (prior to registering with a runtime key) stipulates the conditions and terms under which the Encompass API can be used. Be sure to read and understand these conditions prior to implementing software using the Encompass API.

An *API Runtime license* is a client-side, single-machine, non-transferrable license that permits use of the Encompass API on that machine. In many cases, an API Runtime license is granted implicitly, as is the case for Plugins and Custom Input Form Codebase Assemblies. Developers who author such components do not have to worry about generating runtime licenses on each client desktop – your Encompass license implicitly grants these rights for any machine using Encompass under your license.

On the other hand, developers who author stand-alone applications that load and invoke the EncompassObjects API must ensure that any computer on which their application runs is properly registered with an Encompass API Runtime License. In practice, this means that a distinct, non-transferrable license must be generated on each computer prior to the execution of the custom application.

Generating an API Runtime License starts with obtaining an Encompass API Runtime CD Key. Licensed Encompass customers can request this key from their Encompass Account Representative. Encompass SDK Partners can obtain their license key by contacting their Partner Account Representative.

Once in possession of the API Runtime CD Key, generating a license on a machine can be done in one of two ways:

- If the computer has Encompass installed on it, you can launch the Encompass Admin Tools application and select the “Register Encompass SDK” tool. Enter your company’s API Runtime CD Key and proceed thru the registration process.
- Alternatively, you can build the registration process directly into your custom software. The EncompassObjects API exposes classes under the EllieMae.Encompass.Licensing namespace that allow for self-registration. In particular, you can use the LicenseManager class’s ValidateLicense() method to check for a valid runtime license, and its GenerateLicense() method to generate a new license.

It is important to note that the SDK License Agreement requires that the key used to register a machine be issued to the owner of that system. For example, it is a violation for an SDK Partner to sell software that self-registers using the partner’s API Runtime CD Key. If a partner sells an application to an Encompass licensee, it is the licensee’s API Runtime CD Key that must be used to register the API Runtime on their computers.

Encompass Versioning

Because Encompass is a client-server application, the software enforces a strict versioning policy to ensure the client and server are version-compatible when a connection is made between them.

The versioning policy is very simple. The Encompass software versions running on the client and server must match exactly for the connection to be permitted. Thus, a server running version 16.2 of Encompass will only permit connections from clients running version 16.2 of Encompass or running on top of version 16.2 of the EncompassObjects API.

When a new version of Encompass is released, the Encompass Server is first component upgraded. When a user logs into the Encompass Server using the Encompass SmartClient, the version update is detected and the SmartClient automatically streams the required changes to the user’s computer.

If your application follows the guidelines set forth earlier in this document, your custom application can also leverage the auto-updating capabilities of the Encompass SmartClient. Your application will be “SmartClient-enabled” if the following three conditions are met:

- You have initialized the API Runtime Services in your application prior to using or referencing the EncompassObjects assembly. (See Section 3, Getting Started, for details.)
- You do not have the EncompassObjects assembly or any of its dependencies in the .NET probing path for your application. This includes your application’s bin folder as well as the GAC.
- You do not have the Encompass SDK installed on the computer.

If you have satisfied these three requirements, the Encompass API Runtime Services will handle ensuring that your software is using the correct component versions based on the Encompass Server version. Conversely, if you fail to meet any one of these three requirements, then you will be required to manually upgrade the computer with the correct Encompass components whenever a server version upgrade occurs.

Updating Computers with the Encompass SDK Installed

The most frequent exception that developers will encounter to the three conditions required to be “SmartClient-enabled” is that they have installed the Encompass SDK on the computer. Certainly this is common on a machine intended for use as a development system, but it may also be the case when your application runs on a web server or other back-office system that has no need for a full Encompass SmartClient installation.

In these cases, you will need to be prepared to update the Encompass SDK whenever a server version change occurs. The Encompass SDK installation package includes a tool specifically designed to facilitate this process. The SDK Update Wizard allows you to detect the software version of your Encompass Server and then download and install the appropriate version of the Encompass SDK. You can launch the SDK Update Wizard from the Start Menu.

If you fail to upgrade your SDK when a server update occurs, your custom application will be unable to connect to the Encompass Server. Within your application code, this failure will appear as a VersionException that is raised when you invoke the Session.Start() method. If you are authoring software that will run in environments outside the developer’s control (e.g. software that is re-sold under the Encompass SDK Partner Agreement), it is a best practice to include code to catch this event and notify the user that their software version is incompatible.

For example, you can use the following code in a Windows Form-based C# application.

```
try
{
   Session session = new Session();
   Session.Start("remoteserver", "mary", "maryspwd");
   ...
}
catch (VersionException ex)
{
   MessageBox.Show("The client is unable to connect to the
      Encompass Server because the Encompass software
      versions are incompatible. Please update your Encompass
      software using the SDK Update Wizard located in your
      Start\Encompass SDK menu.");
   }
```

Appendix A

Troubleshooting the EncompassObjects API

Appendix A describes common errors received while using the EncompassObjects API, including the cause of the error and the recommended resolution.

Problem 1

Your application receives the following Exception when invoking the EncompassObjects API.

"The type initializer for 'EllieMae.Encompass.Licensing.LicenseManager' threw an exception"

Additionally, the InnerException property of the Exception object indicates a Permission-related error.

Cause and Resolution

The anonymous user account for your Web application has not been granted local administration rights. Add the anonymous user (IUSR_<MACHINENAME>) to the Administrators group or modify the virtual root's security settings to use an administrator for the anonymous connections to the server. After you assign the user the appropriate rights, you must completely stop and restart the IIS service process by running ***iisreset*** at a command prompt.

Problem 2

Your application receives the following Exception when invoking the EncompassObjects API.

"The type initializer for 'EllieMae.Encompass.Licensing.LicenseManager' threw an exception"

Additionally, the InnerException property of the Exception indicates an "error reading MAC address information."

Cause and Resolution

The most likely cause is that the Windows Management Instrumentation (WMI) system on the computer is corrupt or not running. Ensure that that WMI service is running and that you do not receive any errors when you access the WMI properties thru the Computer Management console. You can also download the WMI Diagnosis Utility from Microsoft's web site to help identify and repair a corrupt WMI database.

Problem 3

You receive the following error when using the EncompassObjects API.

"Could not load file or assembly 'EncompassObjects,...' or one of its dependencies. The system cannot find the file specified."

Cause and Resolution

The EncompassObjects Runtime Services are not being properly initialized or their DLLs have not been deployed to the correct folder. For additional information, refer to the section, "Deploying an EncompassObjects Application" on page 8.

Problem 4

You have written a .NET application that uses the EncompassObjects API and, despite adding a reference to the EncompassObjects and EllieMae.Encompass.Runtime assemblies, you are unable to use those components. Visual Studio indicates that "the type or namespace 'EllieMae' could not be found."

Cause and Resolution

By default, Visual Studio sets the properties of most new applications to use the "Client Profile" portion of the .NET Framework. The EncompassObjects API requires the full .NET 4.5 profile and, as a result, an application that uses only the Client Profile cannot use the Encompass assemblies. Resolve the issue by setting the Target Framework for your project (on the project's Properties screen) to use ".NET Framework 4.5"

Problem 5

You have an existing .NET application written using .NET 3.x or prior, and you are receiving a FileNotFoundException when initializing the Encompass API Runtime Services. Looking in the Windows Event Log, you find the following error:

An error occurred while loading assembly file 'C:\...\EncompassObjects.dll'.

Details: This assembly is built by a runtime newer than the currently loaded runtime and cannot be loaded. (Exception from HRESULT: 0x8013101B)

Cause and Resolution

The Encompass software currently uses .NET 4.5, meaning that all existing applications build on earlier versions of .NET must be either recompiled for .NET 4.5 or include an application configuration file that indicates .NET 4.5 as a <supportedRuntime>. See Section 3, “Installation” on page 6.

Problem 6

Your application that uses the EncompassObjects API fails with an error similar to the following:

“The type initializer for 'EllieMae.Encompass.Client.Session' threw an exception”.

Cause and Resolution

This error is caused when the API attempts to create a log file and does not have permission to write to the directory in which the log is to be created. Whenever the API is initialized, it creates a log file named “Session.log” beneath the directory specified by the DataDir registry value under the HKLM\Software\Ellie Mae\Encompass key. If that key is not present, the log file will be created beneath the %USERPROFILE%\LocalSettings\ApplicationData\Encompass folder. To resolve the issue, ensure that your application has the necessary rights to create files in the specified directory.

The problem is most prevalent if you are running a web application and do not have the full Encompass application installed on your computer. In this case, the DataDir registry value will not exist and the user context under which the application runs may not have access to its own LocalSettings folder. You can add the DataDir value to the registry and set its value to any directory location you wish. Then set the permissions on this directory so the web application's user account has full access.

Problem 7

You receive the error “Unable to connect to remote server. Server may be stopped or refusing connections” when attempting to connect to the Encompass Server through the API. You can connect to the same Encompass Server without problems using the Encompass application on the same computer.

Cause and Resolution

The most likely cause of this problem is that your application is compiled against or running against the incorrect version of .NET. Because the EncompassObjects API requires .NET 4.0, you should ensure that your project is running with this version.

Problem 8

When your application attempts to use the EncompassObjects API you receive a System.BadImageFormatException.

Cause and Resolution

The most likely cause of this problem is that your application is compiled against or running against the incorrect version of .NET. Because the EncompassObjects API requires .NET 4.5, you should ensure that that your project is running with this version.

Problem 9

When your application attempts to use one of the core milestone properties you receive a *Milestone not found* exception.

Cause and Resolution

This exception is caused by the milestone being renamed in the Encompass settings (specifically, the Milestones setting). Starting with Encompass 9.0, there is no longer a concept of a “core” milestone. The “core” Milestone properties are now marked as obsolete and only exist for backward compatibility. *Milestones.GetItemByName(string name)* with the correct name of the milestone should be used instead.

Problem 10

Your application receives the following exception when calling Session.Start().

“Version mismatch attempting to connect to server. Client version = X.X.X.X, Server version X.X.X.X”

Cause and Resolution

This is caused by the SDK program using SDK DLLs that are a different version than the DLLs that are running on the Encompass server that is being connected to. Ensure that the SDK DLL files are up to date with the version of Encompass running on the server.

Problem 11

When your application attempts to get a Milestone value from the Reporting Database using “Fields.MS.*”, a *FieldNotInDBException* exception is triggered along with the following message:

“The field 'MS.*' is not in the reporting database. When trying to add the field to the reporting database it cannot be found.”

Cause and Resolution

This exception is triggered because the MS.* fields are now marked as obsolete. Replace the format “MS.*” with “Log.MS.*”.

For example, if you previously used “Fields.MS.APP” to get the Approval milestone value from the Reporting Database, you should now use “Fields.Log.MS.Approval”.

Problem 12

The following error is returned when attempting to attach a custom codebase assembly DLL to your custom input form: “The Specified Assembly is not a valid .NET assembly or could not be loaded.”

Cause and Resolution

The most likely cause of this problem is that your DLL was compiled in Visual Studio with the 'Target platform' set to *x64*. Encompass runs as a 32-bit application on a 64-bit Operating System. Therefore, 'Target platform' should be set to *ANY CPU*.

Problem 13

When creating an SDK project using VB.NET your application may have a build error that mentions a missing reference to an Encompass DLL. For example, trying to create a new instance of EllieMae.Encompass.BusinessObjects.DataObject will give the following build error:

“Reference required to assembly 'EMCommon, Version=1.5.1.0, Culture=neutral, PublicKeyToken=d11ef57bba4acf91' containing the type 'EllieMae.EMLite.RemotingServices.BinaryObject'. Add one to your project.”

Cause and Resolution

This is caused by differences between the C# and VB.NET compiler. To resolve this issue, add the required references as stated in the error. Like the EncompassObjects assembly, any added Encompass assembly references should have the **Copy Local** option set to *False* and should not be deployed with the project.