CASEMANAGER

Case Manager 5.2 - Instance Directory

Integration Guide

Table of Contents

SECTION 1 OVERVIEW AND PURPOSE

This guide is aimed at developers integrating with the Case Manager technology, specifically the Instance Directory. Most of the concepts surrounding the Instance Directory are covered in the Instance Directory Technical Manual and will not be addressed in this document. A thorough programming understanding and a familiarity with Case Manager concepts are assumed.

Case Manager provides a number of components that simplify integration with the technology. This document discusses a number of these components and includes code examples.

SECTION 2 DEFINITIONS

Instance Directory

The Instance Directory is a network-wide maintained directory of the available named sets of database connection details (instances) for Case Manager applications.

Version control

Version control is the process of ensuring that the database and application are on the appropriate Case Manager version.

XQL

XQL is the VoyagerNetz-defined query language used to communicate with the database. The database connection created by the Database Connection Factory is designed to be used with XQL.

Extension

An extension allows a developer to expand the Case Manager functionality by implementing an interface. The interface defines a number of areas of functionality that can be extended.

Policy

A policy, in terms of the Instance Directory, defines how the application interacts with the Instance Directory - specifically in getting the updated instance information, selecting the appropriate instance from available instances, and behaviour upon failure of initialisation.

Command-line Argument

A command-line argument is a parameter that can be specified when the application is run. It is defined, in a Console environment, by trailing the application execution command with the argument values.

Solution

A Case Manager Solution defines an implementation of the Case Manager technology to address the need of a specific client or market segment.

Licensing Validation

The License Validation process determines the available licenses and the limitations of these licenses of the current system (client) on the relevant technology.

SECTION 3 COMPONENTS

This section hosts descriptions of a number of important components in the Case Manager technology. The purpose of each component, along with an overview of its use and inner workings, is described.

DatabaseConnectionFactory

The `DatabaseConnectionFactory` is a static class used to obtain a database connection from anywhere in the application. During start-up it can be initialised, using information obtained from the Instance Directory, such that all calls to the method `GetConnection()` will return a database connection to the database defined in the instance.

Through properties one can access all the necessary connection details as well as a `ConnectionSetupDescription` that can be used to display information to the user regarding the current instance. Typically this information should be included in the title bar of an application.

The process of initialising the Database Connection Factory is simplified by the use of policies. The policy defines the complete interaction between the Instance Directory and the application. Refer to the Instance Directory Technical Manual for more on policies. The `DatabaseConnectionFactory` class can easily be initialised by calling the `Initialize()` method[1] that takes an Application Interactor and a policy as parameters. An exception will be thrown if the initialisation fails.

IApplicationInteractor

An Application Interactor allows interaction between the user and the application. Different standard implementations are available depending on the type of application:

- **Console**
  All interaction will be done through Console messages and input.
- **Windows Forms**
  Standard .NET Windows Forms Message Boxes are used for interaction between the user and application.
- **Developer Express**
  Message Boxes and forms built on Developer Express technology is used.
- **Windows Service**
  Although user interaction is not supported, communication is done through the Windows Events system.

One simply has to instantiate the appropriate standard Application Interactor in the application. The Application Interactor also defines how an application is restarted. To support the application restart functionality (required by some Instance Directory policies) one has to implement an `Action`, defining the restart process. An example of a restart action can be found in the Restartable application example in this document.

CuminApplication

The `CuminApplication` is a façade that helps with initialisation of the application, specifically:

- Database Initialisation

[1] In most cases the `Initialize` method will not be called directly, as the process of database connection factory initialisation is further simplified by the `CuminApplication`. Only make use of direct initialisation of the `DatabaseConnectionFactory` if the `CuminApplication` is not used.

- Command-line Argument Processing
- Case Manager Version Validation

Basic Operation

After setting up all the parameters of the Cumin Application, the `run()` method will set up the application as specified and pass control to the `Main` method. An additional method, `MainOnEnd`, should be specified that will execute when the application restarts or when it terminates. An **IApplicationInteractor** should also be specified in the Cumin Application that will be used to communicate with the user.

Database Connection Initialisation

The `CuminApplication` façade further simplifies the entire process of Database Connection Factory initialisation allowing the developer to focus only on the application and not the complexities surrounding the database connection and the Instance Directory.

To initialise the Database Connection Factory for the application, simply set `DefaultDatabaseConnectionFactoryInitializePolicy` to an instance of default policy for the application. Passing the command-line arguments to the `run()` method of the `CuminApplication` would process the arguments and allow overriding the default initialisation behaviour.

The `Main` method can communicate an application restart by throwing either an `OperationCancelledException` (as is thrown by `CancellationToken`, discussed later in this document) or an `ApplicationRestartException`. To restart the application and make use of the fail-over process as defined in the application's policy, a `DatabaseFailOverRestartException` should be thrown.

As simple as that! To allow your application to behave correctly during a restart you might need to follow a guideline or two in the `Main` method (which we will discuss in the Restartable application example), but the rest is handled: database connection factory initialisation, error handling and communication, fail over, and policy-related command-line arguments.

Command-line Argument Processing

Command-line arguments related to Database Connection Factory initialisation are parsed and processed by the `CuminApplication`. The façade also allows you to extend the argument processing capabilities: setting the `ArgumentParserExtension` property to your custom implementation of the `ICommandLineArgumentParserExtension` interface will allow the `CuminApplication` to parse and process command-line arguments specific to the application.

All arguments are assumed to be in the format `-name=value` or simply `-name` where *name* is the name of the argument and *value* is the value it is set to. The interface requires the implementation of three properties:

- **ArgumentParser** defines all the argument options and how to process them.
- **ProcessArgumentAction** is an action that can be programmed to do additional processing and verification of the arguments after parsing. This is typically used to verify that all the required arguments were specified by the user.
- **HelpString** is used to communicate help information to the user.

When unrecognised arguments are passed to the application, execution will still continue. By default a warning message will be communicated to the user via the selected output, but can be disabled by setting the `ShouldWarnOnUnrecognisedCommandLineArguments` property of `CuminApplication`.

Case Manager Version Verification

Applications are often written for a specific version of Case Manager to ensure that database structure is what the application expects. The `CuminApplication` supports an easy way of verifying the version of the instance as part of the Database Connection Factory initialisation process. Three modes of verification are supported by setting the `VersionVerificationType` property to the appropriate value:

- **Full**: The full version number of the application should match the database version.
- **DatabaseOnly**: The application version, up to the build number, should match the respective values of the database version. That is, the revision number is ignored.
- **None**: No version checks are performed.

If the version verification fails, the Database Connection Factory initialisation will also fail and trigger the fail-over process of the policy.

Cancel

For restartable applications, `CuminApplication` supports a `Cancel` method which will set the state of the `CancellationToken` passed to the `Main` method to *Cancelled*. Throughout the start-up of the application one should call the `ThrowIfCancellationRequested()` method on the `CancellationToken`, which will throw a `OperationCanceledException` if the state of the token is *Cancelled*. When this exception is not handled by the `Main` method, the fail-over process will be initialised. For a simple example refer to the Restartable application example in this document.

ICommandLineArgumentParserExtension

The `ICommandLineArgumentParserExtension` interface is used to extend the command-line argument parsing functionality of an application. The custom implementation of the interface will be used to process the arguments unrecognised by the standard (Database Connection Factory initialisation) argument parser. The instance of the argument parser extension will be passed as a parameter of the `Main` method such that the processed arguments can be accessed. Three properties are required by the interface:

ArgumentParser

The `ArgumentParser` is a collection of `ArgumentOptions`: pairs of argument names and processing functions. The argument name should be unique for the application – this includes the standard arguments as defined for Database Connection Factory initialisation. The processing function should take the string (the *value* passed for the current argument) as parameter and should return an `ArgumentParseResultEnum` value indicating whether the argument was successfully processed:

- **Successful** indicates that the argument is successfully parsed and processed.
- **Failed** indicates that the argument could not be parsed/processed.
- **Ignored** arguments were successfully parsed but were ignored. For example, when a specific argument is not applicable.

Lambda-functions, as done in the Application-specific command-line arguments example, simplifies setting up the parser extension tremendously.

ProcessArgumentsAction

After all the arguments are processed, additional processing might be required. For example, one might have to verify that all the required arguments are specified. An `Action` is made available through the interface that allows such processing. A `CommandLineArgumentException` should be used to communicate validation failures and errors to the user. When the given action returns normally, successful validation is assumed and execution of the `run()`-method will continue.

HelpString

Lastly, information should be supplied to the user regarding the available command-line arguments and how to use them. The help string provided by the custom-implemented interface will be appended to the standard help information when requested by the user using the `-help` argument.

```
<Custom Application Name> arguments:
argument1 :   Here, a description of the argument and
              possible values should be discussed.
argument2 :   Another argument, another description

If required, some additional information to use the
arguments.
```

CuminSession

The `CuminSession` is singleton class that gives the programmer access to the different Case Manager components. The state of these components represent the session of the current user, including security information, plug-ins, license information, and a range of extendible, generic Session Managers.

Singleton instance

The static property, `Current`, is used to access the singleton instance of the `CuminSession`. Upon restart of the application (more specifically, the `MainOnEnd` method) it is important to reset the state of the `CuminSession` such that the correct information is reloaded. To do this, one can dispose of the singleton instance by calling the static `Dispose()` method. On the topic of the `MainOnEnd` method, it is good practice to log the current user out (refer to UserSecurityManager) before the session is disposed.

```
m_CuminApplication.MainOnEndFunction = (() =>
{
    CuminSession.Current.UserSecurityManager.LogOut();
    CuminSession.Dispose();
}
```

Figure 3: Clearing the CuminSession at the end of the Main method

UserSecurityManager

The `UserSecurityManager` manages all user login and security information. When a log-in is done, the current user's information and security tokens are loaded by this manager. By making use of the `HasAccess(string)` method, the programmer can verify that the current user has the appropriate access rights. Information on the user (and the user's workgroup) can be accessed through the `User` property. Refer to the CuminSessionSetupDescription for options on the log-in process.

WorkgroupManager

Case Manager Workgroups form hierarchies by each specifying a parent workgroup. These tree structures are difficult to build directly from database queries. The `WorkgroupManager`, thus, provides hierarchical information on Workgroups, such as:

- Whether a workgroup is within a specified branch

- All administrative workgroups
- All users' workgroups within a branch
- All team workgroups
- The complete root path (ancestry) of a workgroup

PluginManager

Case Manager operates on an extension framework. Database Known Types, Daily Maintenance Processes, Dashboards, additional Session Managers, Data Mergers, Work Item Actions, and Setting Controls are defined by an extension. Some extensions are not dynamically loaded and define base components required by the application. We refer to these as static plugin definitions. Static plugins are loaded by explicit commands (the method `RegisterStaticPlugin`) in the source code, whereas the `CuminSession` can load all the dynamic extensions, if required by the application. The `PluginManager` is used to access all the loaded plugin definitions (static as well as dynamic extensions). Note that all the components of the extension will be registered to the relevant Case Manager functionality automatically (e.g. Database Known Types will be registered to the `DatabaseConnectionFactory` automatically). It is therefore uncommon for an extension or application to access the loaded plugins directly. Extension-specific information would rather be encapsulated in a Session Manager defined in the extension.

SolutionManager

The `SolutionManager` is loaded before login and provides information on the solution implemented in Case Manager. The license may enforce a specific solution to be implemented, based on the license verification type (specified in the CuminSessionSetupDescription). The Solution information defines the branded form background images (the login form, about form and the start-up splash screen), the branding colour, the version of the solution, and limitations of the solution.

LicensingManager

The Case Manager License can be verified using the `LicensingManager`. Based on the verification mode, the license information will be updated from the Case Manager Licensing Service and verified for validity. A license may also require a specific solution to be implemented on the system. The verification process also includes the renewal process of licenses which requires an internet connection and user interaction.

SystemInformation

The `SystemInformation` provides the Client System ID (also referred to as the SGUID) and the client name. This is the same information as provided by the `VoyagerSettings` Session Manager, but is always loaded at the beginning of setup (whether or not the `VoyagerSettings` Session Manager is included in the setup).

SessionManagers

All Session Managers, whether from static plugin definitions or from dynamic extensions, are loaded in the `CuminSession`. The programmer can easily access the appropriate Session Manager by making use of the generic method, `Manager<T>()`. Session Managers are unique in type – that is, only one Session Manager of a type may be registered. Access, using the `Manager` method, is done by requesting the appropriate Session Manager type. For example, if I need access to the `VoyagerSettings` Session Manager, the following method call will result in the registered object of type `VoyagerSettings`:

```
CuminSession.Current.Manager<VoyagerSettings>()
```

Figure 4: Accessing Session Managers in CuminSession

No typecasting is necessary as the object is returned in the type requested. Especially working with dynamic extensions, the programmer often needs to first verify that the Session Manager is loaded. The `IsManagerRegistered<T>()` method may be used for this. The `CuminSessionSetupDescription` defines the standard Session Manager to load – that is, Session Managers defined in the base Case Manager code. Session Managers defined in dynamic extensions (as specified in the plugin definition) will automatically be loaded in the `CuminSession` upon setup.
Since Session Managers are usually defined in external code, it provides an easy mechanism for extensions to maintain their static session information in the `CuminSession`. The `Manager<T>()` method is often used to access different Case Manager components and dynamic extension information.

CuminSessionSetupDescription

Case Manager provides an easy façade to set up the Cumin Session. All the information required for setup is specified in a `CuminSessionSetupDescription` object and passed to the `Setup` method of the `CuminSession` at start-up of the application. The following properties should be set:

ApplicationInteractor

The `ApplicationInteractor` is specified as a parameter to the constructor of the `CuminSessionSetupDescription` and is used for all interaction between the user and the application during setup.

SetupFailureMode

Two modes are supported to handle failure during the setup of the `CuminSession`:

- **Exceptions**: An exception is thrown (the default behaviour).
- **ApplicationInteractor**: An error message is communicated via the Application Interactor and `false` is returned by the `Setup` method.

ShouldLoadDynamicExtensions

This Boolean property indicates whether dynamic extensions should be loaded by the application. When extensions are not required by the application, this value should be `false` (the default value) for the purpose of performance and stability.

ShouldLoadWorkgroupManager

When hierarchical workgroup information is required by the application and the `WorkgroupManager` is used, the `ShouldLoadWorkgroupManager` property should be set to `true`. By default the `WorkgroupManager` would not be loaded.

LicenseApplicationId

The application should be identified for license verification. The GUID specified in this property should be the same as that of the appropriate license in the Case Manager Licensing Service. This property is required if any form of License verification is done (see LicenseValidateMode).

LicenseValidateMode

License validation can be done during setup of the `CuminSession`. There are four modes that can be specified:

- **None**: No license information will be loaded into the Licensing Manager and no verification will be done. This is the default mode.
- **LoadApplicationLicenseOnly:** The application license information is loaded from the database (not from the Licensing Service) but no validation is done.
- **ValidateApplicationLicense:** The application license information is loaded from the database and validated. The process includes the renewal process that will run upon expiry and when the expiry

date is within a warning period. Administrators will be notified if expiry is within the next week and all users will be notified if expiry is within the next day.

- **ValidateApplicationAndSolutionAsExternalIdentifier:** If the application specifies a solution identifier (GUID) as external identifier, this mode can be used to verify that the correct solution is implemented in the system. If the license specifies no external identifier, any solution will be seen as valid. The solution is validated after the normal license validation process.

UserLoginMode

The programmer can specify that a login is required by the application. The Application Interactor already caters for login interaction with the user. The user will be prompted to log in, simply by specifying the appropriate login mode:

- **Standard:** The user is prompted with a login screen where the user and password should be specified. This is the default mode.
- **PasswordForPreSpecifiedUser:** The programmer may specify the user that is to be used by the application (that is, the user does not have the option of another user). The user's password is still required.
- **PreSpecifiedUser:** The programmer specifies the user to be used by the application. No password is required from the user.

SpecificLoginUserId

The programmer should specify the user to be used by the application in the cases where `UserLoginMode` is set to `PasswordForPreSpecifiedUser` and `PreSpecifiedUser`. This is done by specifying the User ID (or USERGUID).

SessionManagerTypes

Session Managers are not added to the `CuminSessionSetupDescription` by using properties. `SessionManagerSetupDescription` object should be added using the `AddSessionManagerSetupDescription` or `AddSessionManagerSetupDescriptionRange` method. The `SessionManagerSetupDescription` class provides static methods to easily create the objects from Session Manager types. For example, creating a `SessionManagerSetupDescription` for `VoyagerSettings` is done using the following statement:

```
SessionManagerSetupDescription.New<VoyagerSettings>()
```

Figure 5: Creating a Session Manager object from type

The `New` method is overloaded to support specifying the Session Manager concrete type but mapped to an implemented interface type, and also specifying the actual instantiated Session Manager object.
Only the Session Managers from the standard Case Manager components should be specified. All the Session Managers from the dynamic extensions will be loaded automatically if the `ShouldLoadDynamicExtensions` is `true`.

ProgressObservers

Progress Observers (implements the interface `IProgressObserver`) can be added to the CuminSessionSetupDescription (through the method `AddProgressObserver`) to communicate setup progress. The Splash screen used by Case Manager is an example of observing the setup progress.

BaseCuminPlugin

The fundamental components as defined in the static `BaseCuminPlugin` are:

- All the Case Manager object known types
- The Base Daily Maintenance Process
- The “Execute Instructions” Daily Maintenance Process
- The activity, status and timing dashboards (both Wall Dashboards and Windows Dashboards)
- The Tag Merger (used with short message merging) for all the standard Case Manager objects related to a case.
- Initialisation of
    - The Legacy User Security Settings
    - The Custom (and Case Extra) Field Names
    - The SMS Service Client
    - The Print Manager

It is therefore important to load the plugin definition statically at the start of the program - before the `CuminSession` is set up:

```
CuminSession.Current.PluginManager.RegisterStaticPlugin(
                                        new BaseCuminPlugin());
```

Figure 6: Registering the Base Cumin Plugin

ICancellable

The `ICancellable` interface provides a mechanism to cancel processes (especially forms) through an external call. This can be used to implement a restart function (in the Application Interactor) for an application that makes use of forms. Refer to the Restartable application for an example of the use of the interface.

Case Manager Components

WorkItemMediator

The `WorkItemMediator` handles all communication when work items (cases) are opened, closed, gain focus or change state. Colleagues to the `WorkItemMediator` are notified when any of these actions occur, to which they may respond appropriately. An example of this is the Case Manager Front End (through the `CuminDisplayManager`) and the Timer. In general when dealing with instructions and activities, the `WorkItemMediator` becomes necessary.

VoyagerSettings

The `VoyagerSettings` stores the basic settings of Case Manager. This includes many of the financial and system-wide settings. In order to access these settings, one has to specify the `VoyagerSettings` as a Session Manager. Requesting the manager through the `CuminSettings` will give the programmer access to all the settings as stored for the system.

ThemeSkinManager

Case Manager provides a very easy mechanism of using the form skins – standard skins as well as the themed skins – in a custom application. When the `ThemeSkinManager` is added to a `CuminSession`, all Developer Express forms will use the appropriate skin. The themed skins can be deactivated by setting the `IsThemesEnabled` property to `false`. The only further interaction with this Session Manager is when a form is created in another thread (which will not, by default, take on the original skin). The following call in the new thread will resolve the problem:

```
if (CuminSession.Current.IsManagerRegistered<ThemeSkinManager>())
{
    CuminSession.Current.Manager<ThemeSkinManager>().
                                .SetThemeForCurrentEnvironment();
}
```

Figure 7: Applying themed skins in another thread

LicenseSessionManager

When a license stipulates a specific amount of available licenses, it can be enforced by an application through the `LicenseSessionManager`. The license specified as the Application License (see `LicenseApplicationId`) will be used. A check-out is performed: an entry for the user on the specific workstation will be logged for this application license. If the amount of checked out licenses exceeds the amount specified by the license, the application will not initialise and an appropriate message will be displayed to the user. To simplify, by adding the `LicenseSessionManager` to your application (given that the amount of licenses is specifically for your application), the amount of users will be enforced.

ElectronicDiary

The `ElectronicDiary` provides information on work items assigned to users and workgroups. Only when this information is required should this Session Manager be added to the application as there is a rather severe implication on the setup speed.

FeedbackReportClient

A web service is available for communication of errors and user feedback from Case Manager software. If you choose to include this functionality in your application, simply add the `FeedbackReportClient` Session Manager to your application and make use of the `SendFeedback` method.

SECTION 4 EXAMPLE PROGRAMS

Simple Instance Directory integrated application

This simple example shows the general structure of using the `CuminApplication`. The program entry point - `Main(string[])` - sets up and runs the `CuminApplication` set to call another `Main` method (taking an `ICommandLineArgumentParserExtension` and a `CancellationToken` as arguments) with an already set up `DatabaseConnectionFactory`.

Application-specific command-line arguments

Let's create a simple application that takes the following arguments:

- **FilterType**: either "LastName" or "MinBalance"
- **LastName**: the last name to be used in the filter
- **Balance**: the minimum balance applied as a filter

When the filter type is specified to be `LastName`, the `LastName` argument must be set. Similarly, when `MinBalance` is specified, the `Balance` argument should be set. A filter must be specified.
We will work from the Simple Instance Directory integrated application example.

Create a new Command-line argument parser extension:

```
internal ExampleArgParserExtension()
{
    this.ArgumentPraser = new ArgumentParser();
    this.ArgumentParser.AddOption(new ArgumentOption("FilterType", (value) =>
    {
        if (String.IsNullOrWhiteSpace(value))
        {
            return ArgumentParser.ArgumentParseResultEnum.Failed;
        }
        else if (value.Equals("LastName", StringComparison.OrdinalIgnoreCase))
        {
            this.FilterType = ExampleFilterTypeEnum.LastName;
            return ArgumentParser.ArgumentParseResultEnum.Successful;
        }
        else if (value.Equals("MinBalance", StringComparison.OrdinalIgnoreCase))
        {
            this.FilterType = ExampleFilterTypeEnum.MinBalance;
            return ArgumentParser.ArgumentParseResultEnum.Successful;
        }
        else
        {
            return ArgumentParser.ArgumentParseResultEnum.Failed;
        }
    }));

    this.ArgumentParser.AddOption(new ArgumentOption("LastName", (value) =>
    {
        this.LastName = value;
        return String.IsNullOrWhiteSpace(value)
                ?  ArgumentParser.ArgumentParseResultEnum.Failed
                :  ArgumentParser.ArgumentParseResultEnum.Successful;
    }));

    this.ArgumentParser.AddOption(new ArgumentOption("Balance", (value) =>
    {
        if (String.IsNullOfWhiteSpace(value))
        {
            return ArgumentParser.ArgumentParseResultEnum.Failed;
        }
        else
        {
            double balance = 0;
            if (Double.TryParse(value, out balance))
```

```
            this.ProcessArgumentsAction = () =>
            {
                if (this.FilterType == ExampleFilterTypeEnum.LastName &&
                    String.IsNullOrWhiteSpace(this.LastName))
                {
                    throw new CommandLineArgumentException("Last name not specified");
                }

                if (this.FilterType == ExampleFilterTypeEnum.MinBalance &&
                    !this.Balance.HasValue)
                {
                    throw new CommandLineArgumentException("Balance not specified");
                }
            };
        }
    }
```

Figure 9: Creating an argument parser extension

Changes to the CuminApplication

The `CuminApplication` will process your application's command-line arguments appropriately when the `ArgumentParserExtension` property is set to an instance of your application's custom argument parser extension. The arguments passed to the `Main` method should, of course, be passed to the `Run` method.

```
m_CuminApplication.ArgumentParserExtension = new ExampleArgParserExtension();

...
m_CuminApplication.Run(args);
```

Figure 10: Registering an argument parser extension

Using parsed arguments in Main method

The instance of your application's argument parser extension – with the arguments parsed – will be passed as an argument to the `Main` method. Thus, simply by casting the object to the appropriate type, you will be able to access the parsed arguments.

```
static void Main(ICommandLineArgumentParserExtension argumentParser,
                 CancellationToken cancellationToken)
{
    ExampleArgParserExtension exampleArgParserExtension =
                  (ExampleArgParserExtension)argumentParser;

    //Can use parsed argument e.g. exampleArgParserExtension.FilterType
    ...
}
```

Figure 11: Access custom parsed arguments in the Main method

Application with Login

The `CuminSession` is used for user login. The base plugin is registered, and the `CuminSession` is initialised with the login mode appropriately set. The following example reflects the necessary changes to the Simple Instance Directory integrated application:

Figure 12: Application with Login

Restartable application

Restartable policies are only supported in applications that specify an Application Interactor with a restart function. The restart function should cancel the `CuminApplication` and request cancellation on all forms (through the `ICancellable` interface). Furthermore, the `CuminSession` should be reset which can be done by logging out the user and disposing the Singleton instance. Let's look at each of the areas affected:

Forms

Main forms should be made cancellable though the `ICancellable` interface. This will allow the form to be closed via an external call. Since the call is generally made by a separate thread, we need to cater for Cross-threading.

Figure 13: Implementing cancellable forms

Application Interactor

For simplicity, let us implement a private helper method to create an Application Interactor to be used by the application. This method should be used to set the `ApplicationInteractor` property of the `CuminApplication`. For this example, we will use the standard Application Interactor for the Developer Express forms; standard Application Interactors are available for Windows Forms, Console Applications and Windows Services as well.

When forms (or other cancellable items are used in the application we need to maintain a list of these items.

```
private static List<ICancellable> m_CancellableItems;
```

Figure 14: Restartable Application Interactor

CuminApplication

The `CuminSession` should be reset when a restart is requested. We can specify this as part of the `MainOnEnd` method (that is, when the `CuminSession` is used in the application):

```
public static void Main(string[] args)
{
    ...
    m_CuminApplication.MainOnEndFunction = (() =>
    {
        CuminSession.Current.UserSecurityManager.LogOut();
        CuminSession.Dispose();
    });
    ...
}
```

Changes to Main

In our `Main` method, we should make regular calls during start-up to check the state of the cancellation token which will throw an `OperationCanceledException` when cancelled. We should not handle the `OperationCanceledException`, `DatabaseFailOverRestartException` and the `ApplicationRestartException` in the `Main` method to ensure the fail-over process initiates correctly.

Cancellable items, typically forms, are added to a static list such that the restart function has access to cancel the item when necessary. The item should also be removed from the list when closed (cancelled) normally.

In this example program, general exceptions are handled when compiled in Release mode only. In Debug mode, exceptions are communicated in Visual Studio much better when unhandled.

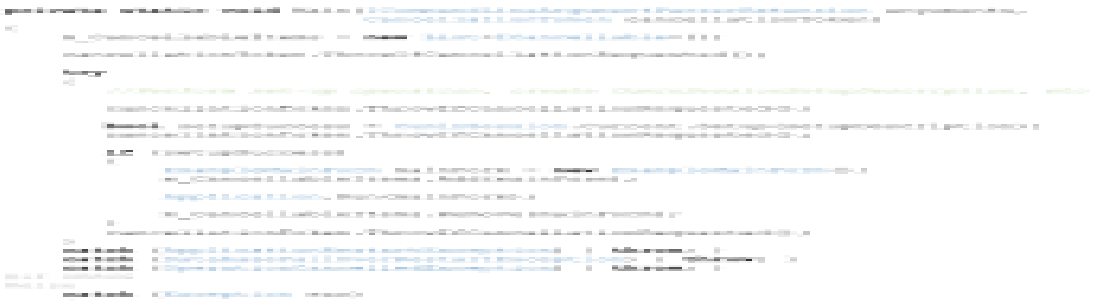

Figure 15: Restartable Main method

License Validated Application

To validate an application's license, simply set the `LicenseValidateMode` and specify the license's application identifier as defined on the Case Manager Licensing Service.

```
private static void Main(ICommandLineArgumentParserExtension arguments,
                         CancellationToken cancellationToken)
{
    ...
    setupDescription.LicenseValidateMode =
        CuminLicensingManager.LicenseValidateModeEnum.ValidateApplicationLicense;
    setupDescription.LicensingApplicationId =
        "{90C5AE56-8149-479A-AB8E-5E1ED6B72891}"; //as on VN License Service
    ...
}
```

Figure 16: Validating an application's license

Application making use of extensions

To make use of extensions in your applications, simply set the `ShouldLoadDynamicExtensions` property to `true`. Session Managers specified in the extension definition will be loaded automatically. You should not specify it directly.

```
setupDescription.ShouldLoadDynamicExtensions = true;
```

Figure 17: Loading Dynamic Extensions

Application making use of additional session managers

When adding standard Session Managers to the setup description, make use of the generic method, `New`. The method is overloaded to take the Session Manager type (if a default constructor exists), a base and a concrete Session Manager type, and a base type along with an actual Session Manager object. The last is specifically used when a non-default constructor should be used.

```
setupDescription.AddSessionManagerSetupDescriptionRange(
   new SessionManagerSetupDescription[]
   {
      SessionManagerSetupDescription.New<ThemeSkinManager>(),
      SessionManagerSetupDescription.New<VoyagerSettings>(),
      SessionManagerSetupDescription.New<ICustomizationManager,
                                         CustomizationManager>(),
      SessionManagerSetupDescription.New<IDisplayManager>(
                                         new SomeDisplayManager(someArgument));
   });
```

Figure 18: Adding Session Managers to the CuminSession

SECTION 5 LIBRARIES

VoyagerNetz.Instance

`VoyagerNetz.Instance` contains all the interaction, policies, and behaviour of the Instance Directory. This library contains the `DatabaseConnectionFactory`, the definition of the `ApplicationInteractor`, and the `ICancellable` interface.

VoyagerNetz.InstanceDirectory

For communication with the remote Instance Directory, common objects are defined in the `VoyagerNetz.InstanceDirectory` library.

Application Interactors

A few standard implementations of the Application Interactor are available.

VoyagerNetz.Instance.DevEx

The standard Application Interactor for Developer Express applications.

VoyagerNetz.Instance.WinForms

The standard Application Interactor for .NET Windows Forms applications.

VoyagerNetz.Instance.Console

The standard Application Interactor for Console applications.

VoyagerNetz.Instance.WinService

The standard Application Interactor for Windows Services.

Microworks.Data

The fundamental library for XQL.

Micwoworks.Data.FirebirdClient

The Firebird implementation of the XQL library.

Microworks.Voyager.Cumin.Common

This library contains all the common Cumin types.

Microworks.Voyager.Cumin.Voyager

All the Case Manager database/Cumin objects are defined in the `Microworks.Voyager.Cumin.Voyager` library.

VoyagerNetz.BaseComponents

The `BaseComponents` library contains the `CuminApplication` and the `CuminSession`. The standard managers of the `CuminSession` - the Plugin Manager, Solution Manager, License Manager, and Version Manager – are also defined in this library. `BaseComponents` also defines the Tag Data Merger, used to merge Short Message Templates.

VoyagerNetz.Components.Work

The functionality related to instructions, action buttons and workflow is defined in the `VoyagerNetz.Components.Work` library.

VoyagerNetz.Licensing.Common

The licensing renewal process includes a file-based sub-process of which the necessary common objects are defined in the `VoyagerNetz.Licensing.Common` library.