Connectivity Toolkit
Reference Manual

September 2005

TOWER Software
Asia-Pacific
www.towersoft.com.au

TOWER Software
Europe, Middle East & Africa
www.towersoft.co.uk

TOWER Software
North America
www.towersoft.com

w w w . t o w e r s o f t . c o m

12012 Sunset Hills Road
Two Discovery Square
Suite 510
Reston, VA 20190
Telephone: 1-800-255-9914
Telephone: +1 (703) 476-4203
Facsimile: +1 (703) 437-9006
E-mail: info@towersoft.com
http://www.towersoft.com/

TOWER Software
Oaklands Park
Wokingham
RG41 2FD
United Kingdom
Telephone +44 (0)118 977 1212
Fax +44 (0)118 979 5444
E-Mail: info@towersoft.co.uk
http://www.towersoft.co.uk

TOWER Software
1st Floor
220 Northbourne Avenue
Braddon ACT 2612
Help Desk: +61 (0) 2 6245 2150
Inquiries: +61 (0) 2 6245 2100
Facsimile: +61 (0) 2 6245 2111
E-Mail: info@towersoft.com.au
http://www.towersoft.com.au

Disclaimer

TOWER Software accepts no liability or responsibility for consequences arising from the use of this product. TOWER Software shall not be liable for any loss howsoever caused whether due to negligence or otherwise arising from the creation, maintenance, or use of any database and this documentation.

Copyright

Trademarks

TRIM and TRIM Context are trademarks of TOWER Software. Other trademarks used herein are used for reference purposes only and are trademarks of their respective owners.

Contents

1. Which SDK?

Introduction

With the release of the TRIM Connectivity Toolkit, it is becoming harder for customers to decide which version of the TRIM Context SDK to be using for their own development efforts. Essentially, TRIM customers are now spoilt for choice in the SDK realm. This chapter seeks to clarify the selection process for SDK versions, and is written from the perspective of a customer seeking to decide on a version of the TRIM Context SDK before starting development work.

What TRIM Context SDK versions are available?

Before being able to discuss which version of the TRIM Context SDK should be used in a given situation, it is worthwhile to reiterate the versions of the TRIM Context SDK which are currently, available. These are:

The TRIM Context COM SDK: this is the traditional TRIM Context SDK option, and is the only one which has been available for the entire life of the TRIM Context product. There is a significant amount of code developed using this SDK version, both internally within TOWER, and with external customers. This SDK makes a series of COM objects available to COM capable platforms and languages – in other words Microsoft Windows centric development environments.

Customer generated Secondary Interop Assemblies (SIAs) (for Microsoft .NET): with the release of the first Microsoft .NET development environments, Microsoft made it possible for developers to reference COM SDKs within the new .NET languages such as C# and VB.NET. This is done by Visual Studio generating an interoperability assembly, which is termed an SIA. These assemblies are generated by each individual developer, and are therefore specific to the version of TRIM and Visual Studio on the developer's machine at the time that the assembly was generated. Whilst TOWER Software has tested SIAs internally, we cannot guarantee that the version of Visual Studio at the customer site produces code the same as that which TOWER Software has tested. SIAs are not signed, and therefore cannot be inserted into the .NET Global Assembly Cache (GAC). A copy of these SIAs must therefore be distributed with each copy of each application which relies upon them. This can result in many different versions of various SIAs being installed on a given customer machine at any one time.

Primary Interop Assemblies (PIAs) (for Microsoft .NET): in recognition of the issues related to SIAs, TOWER Software generated a set of these interoperability assemblies, and released them with TRIM 5.2 SP 2. These are functionally identical to the SIAs that customer developers have generated, except that TOWER Software has tested them, and signed the assemblies as being trustworthy. As the assemblies are signed they may be inserted into the GAC, which means that there only needs to be one copy on any given machine.

TRIM Connectivity Toolkit: TOWER Software has implemented a Web Service version of the functionality available in the TRIM Context SDK. This Web Service relies heavily on established Internet standards such as HTTP, SOAP, and XML, and should therefore be accessible to all languages capable of referencing such a standards compliant Web Service.

Making the selection

What TRIM Context SDK version is right for a given task? The following points will guide you through the decision making process.

You have existing code using the TRIM Context COM SDK – continue to use the COM SDK version. TOWER Software will continue to support this version of the SDK for the foreseeable future.

You are using a language which cannot access .NET but can access COM objects and TRIM is installed on the same machine as the application will run on – use the COM SDK version. The COM SDK provides a richer more interactive interface than the Web Service, and performs well when installed on the same machine as the SDK application.

You have existing code using a SIA – upgrade to the TOWER Software PIAs.

You are writing new code with a .NET language, and have TRIM installed on the same machine as your application will run on – use the TOWER Software PIAs.

You are using a language which supports Web Services based on HTTP, SOAP and XML, and your application will run on a machine without TRIM installed on it – use the TRIM Web Service when it is released, as it is optimized for remote network use. You are using a language which doesn't support COM, or .NET, but which is capable of using Web Services based on HTTP, SOAP, and XML – use the TRIM Web Service when it is released.

Your language doesn't support COM, .NET, or Web Services – unfortunately your chosen language is not capable of using the TRIM SDK.

Conclusion

Hopefully this chapter has helped you decide which TRIM SDK is right for you. The rest of this document assumes that you make a decision to use the TRIM Connectivity Toolkit, and documents how to use the Web Service provided by the Toolkit.

2. Installation

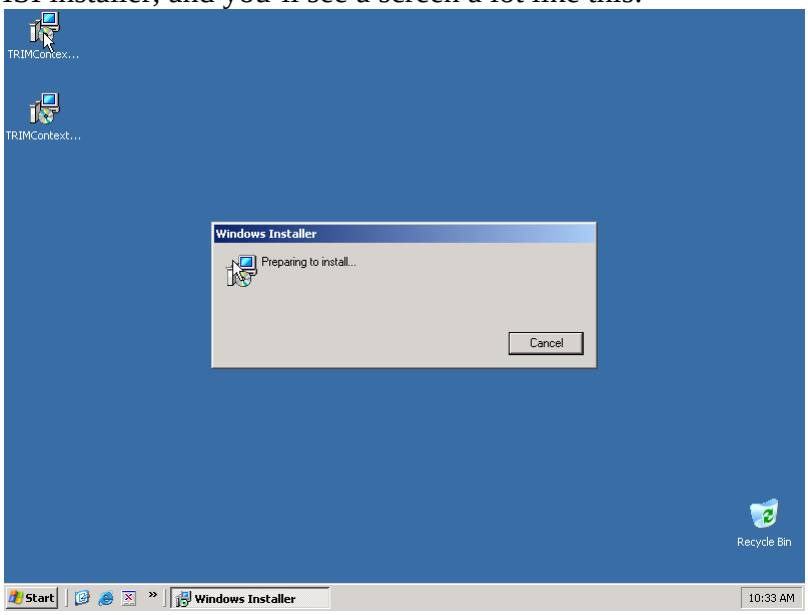
Introduction

This chapter discusses issues around the installation and configuration of the TRIM Connectivity Toolkit. The chapter starts by discussing how to run the MSI installer, and the supported platform for the TRIM Connectivity Toolkit. It then discusses manual deployment of the TRIM Connectivity Toolkit, before finishing up with answers to commonly asked questions.

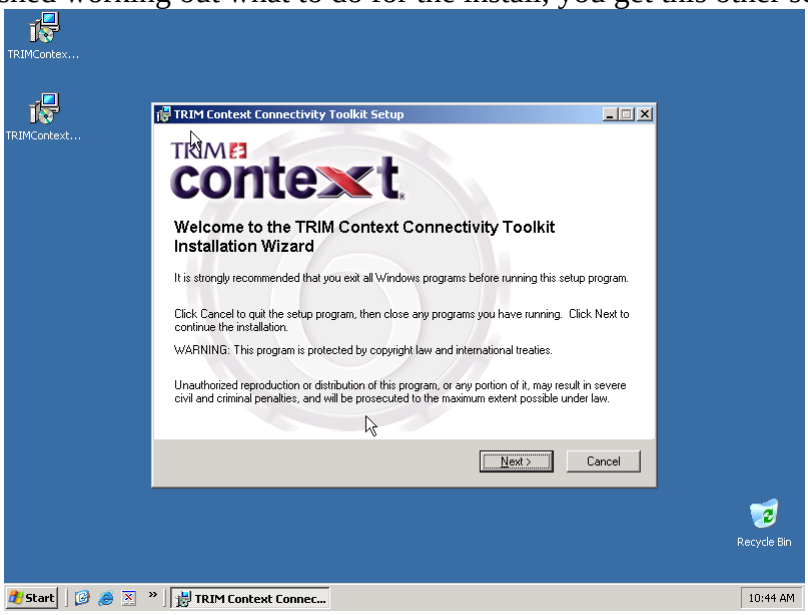
Running the MSI installer

The default method of installation for the TRIM Connectivity Toolkit is to use the MSI installer provided by TOWER Software. Before you run the MSI installer, you need to have IIS 6 installed on the machine. Do this by using the Computer Management applet, or the Add and Remove Programs applet. Once you've installed IIS 6, then you're ready to run the MSI.

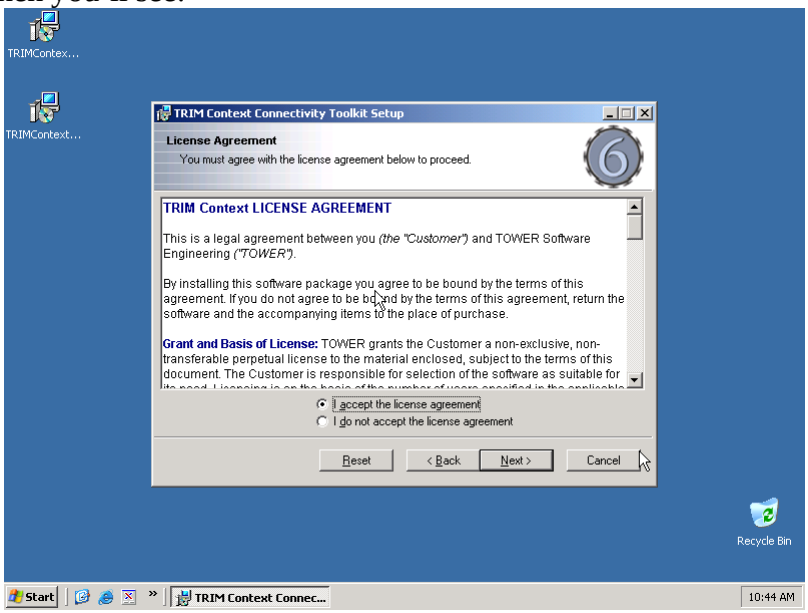
Start up the MSI installer, and you'll see a screen a lot like this:



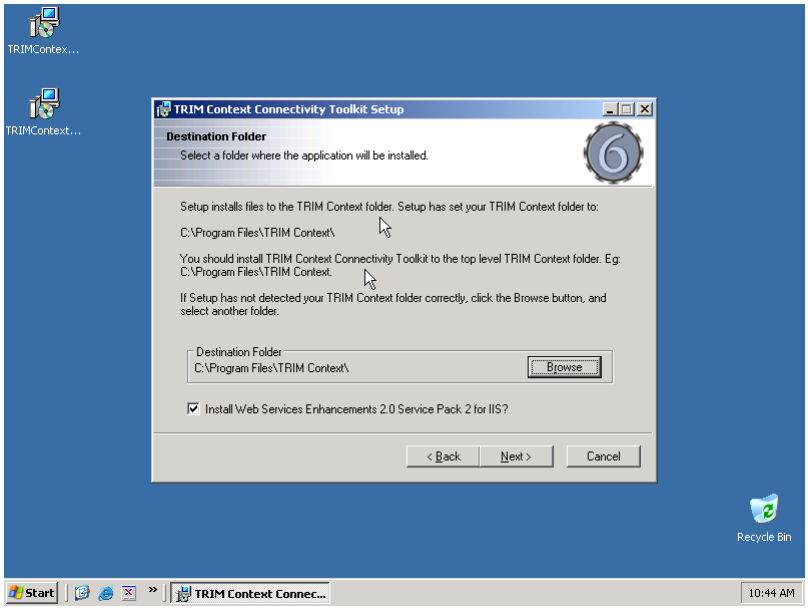
Once it's finished working out what to do for the install, you get this other screen:



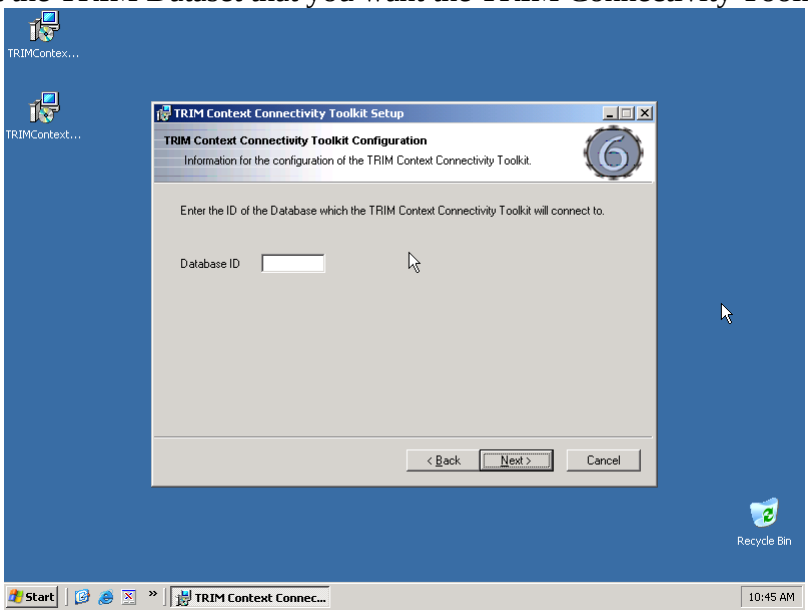
Click next. Then you'll see:



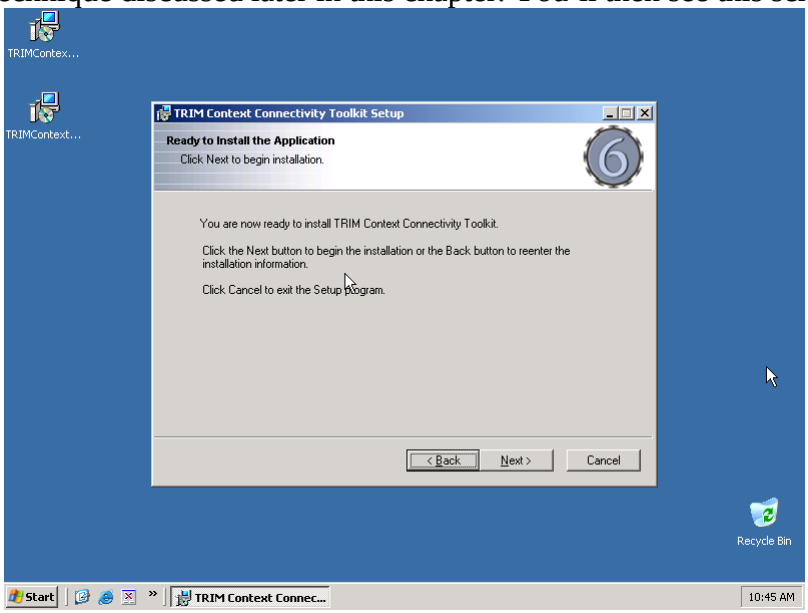
You need to accept TOWER's license agreement for the installer to continue. You'll then see this screen:



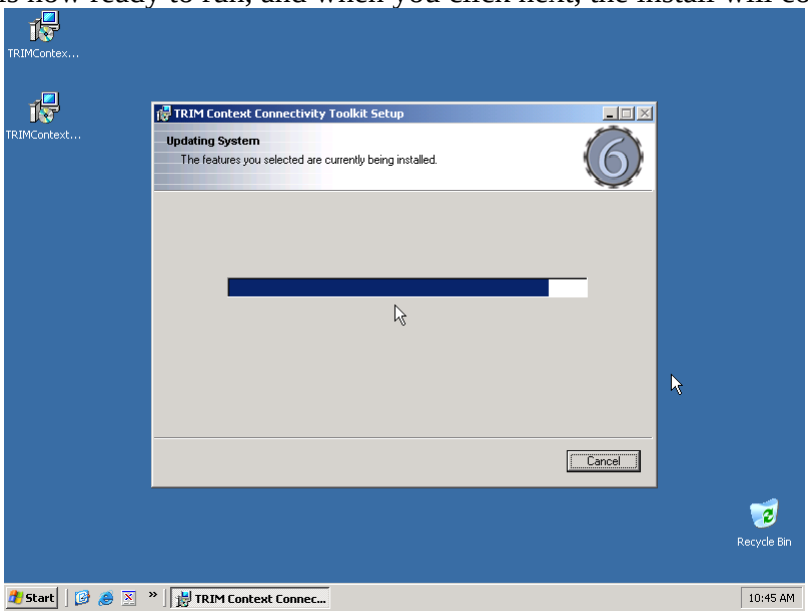
The directory specified here is generally the location of the TRIM installation. This location will be used as the parent directory for the virtual directory that the TRIM Connectivity Toolkit installer will create. This virtual directory is configured within Microsoft IIS 6, and is where the TRIM Connectivity Toolkit binaries reside. Next you need to select the TRIM Dataset that you want the TRIM Connectivity Toolkit to expose:



The TRIM Connectivity Toolkit can only expose one dataset at a time. If you want to support more than one TRIM Dataset via the TRIM Connectivity Toolkit, then you'll need to install two copies of the TRIM Connectivity Toolkit, using the manual deployment technique discussed later in this chapter. You'll then see this screen:



The installer is now ready to run, and when you click next, the install will commence.



The installer reports on progress of the install, and when complete shows this final window:

You are now ready to use the TRIM Connectivity Toolkit.

Supported platform

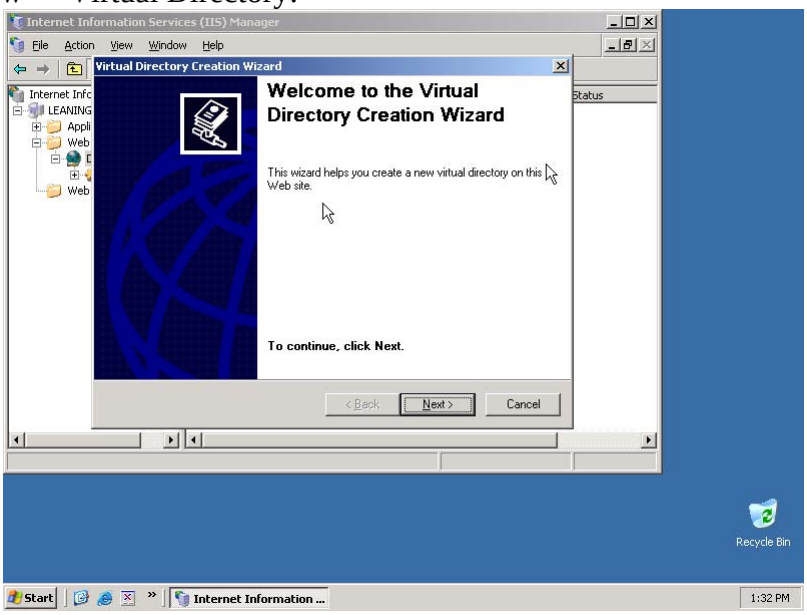
The only platform supported by TOWER Software for the TRIM Connectivity Toolkit is Microsoft Windows Server 2003, running Microsoft IIS 6. Microsoft IIS 6 is included for free with Microsoft Windows Server 2003.

Additionally, you need to have Microsoft's Web Service Extensions 2.0 SP3 installed. This supplies the implementations of the WS-Extension specifications needed by the TRIM Connectivity Toolkit. The TRIM Connectivity Toolkit MSI installed WSE 2.0 SP3 for you if it isn't already installed.

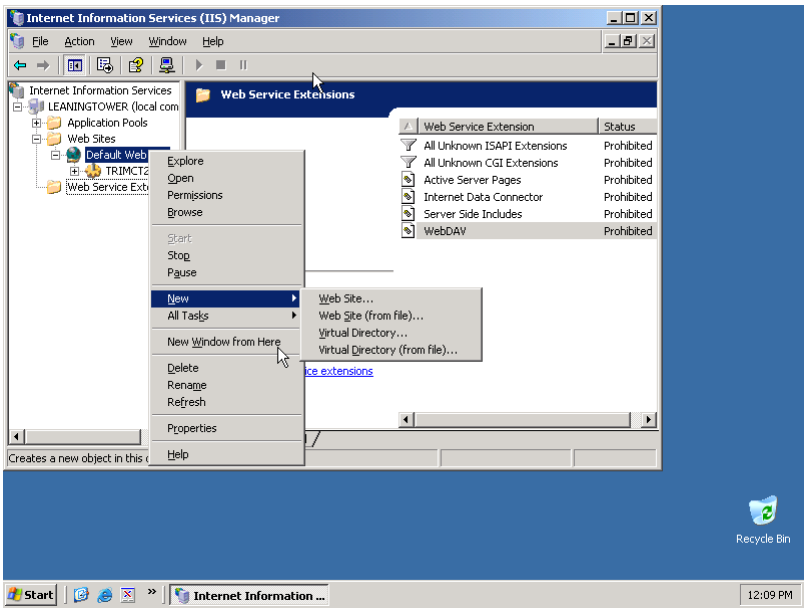
Manual deployment

It is possible to deploy the TRIM Connectivity Toolkit without running the MSI installer. This will often be of use if the TRIM Connectivity Toolkit is already installed on a machine, and you want to run a second instance of the Toolkit on that machine, perhaps pointing to a different TRIM dataset. It is also useful for deploying the TRIM Connectivity Toolkit to a large number of machines in a clustered web environment. The TRIM Connectivity Toolkit runs inside a virtual directory managed by Microsoft's IIS. This virtual directory needs some specific permissions configured, which are normally setup by the MSI installer. To manually deploy the TRIM Connectivity Toolkit, then you'll need to perform these operations yourself.

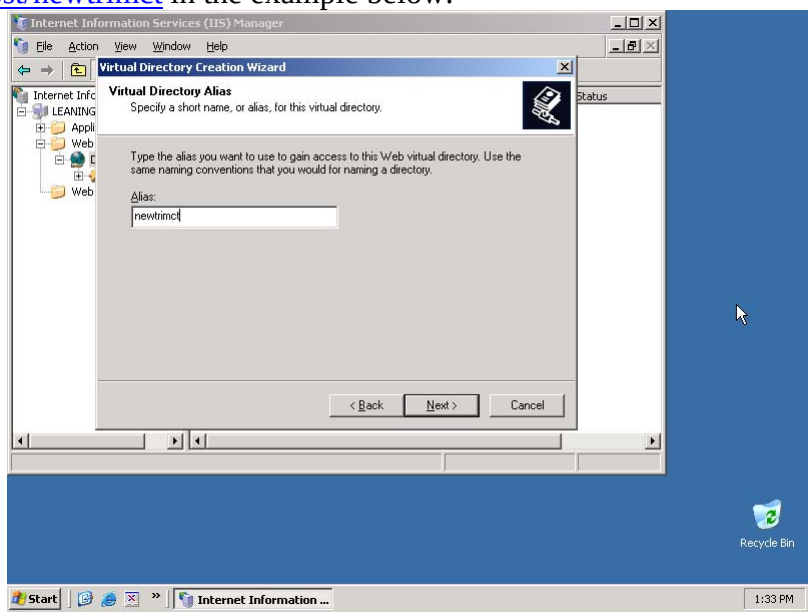
Start the Internet Information Services Management MMC applet. Select the web site for which you want to configure the TRIM Connectivity Toolkit, right click on that web site and select New -> Virtual Directory:



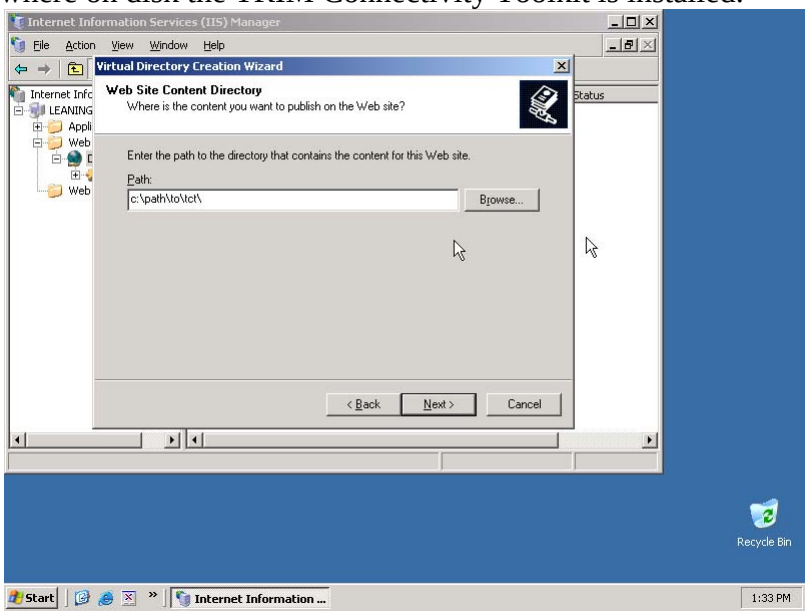
Click next.



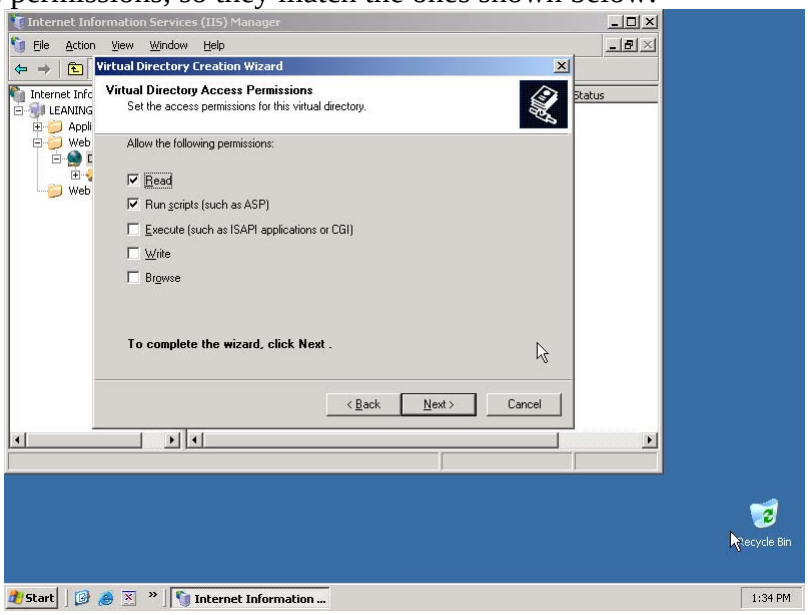
Enter a name for the virtual directory – this will be the name in the URL, for instance http://localhost/newtrimct in the example below:



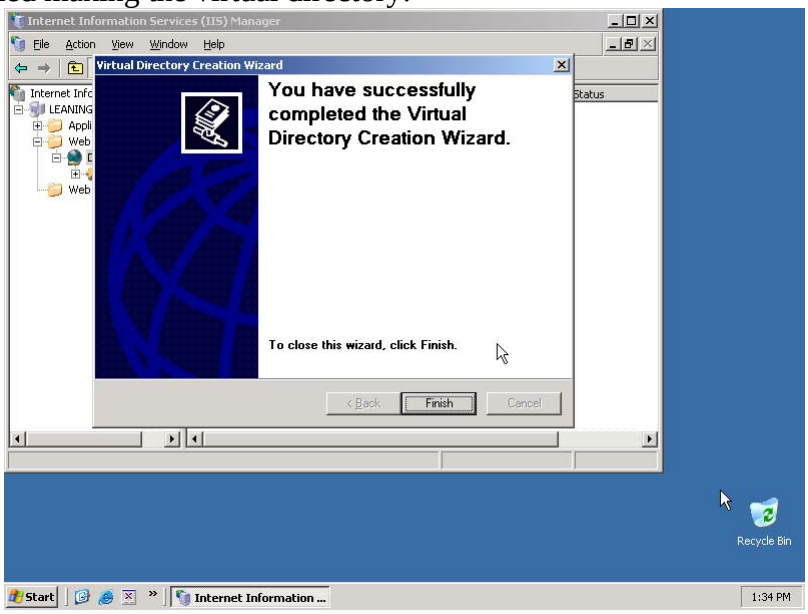
Now specify where on disk the TRIM Connectivity Toolkit is installed:



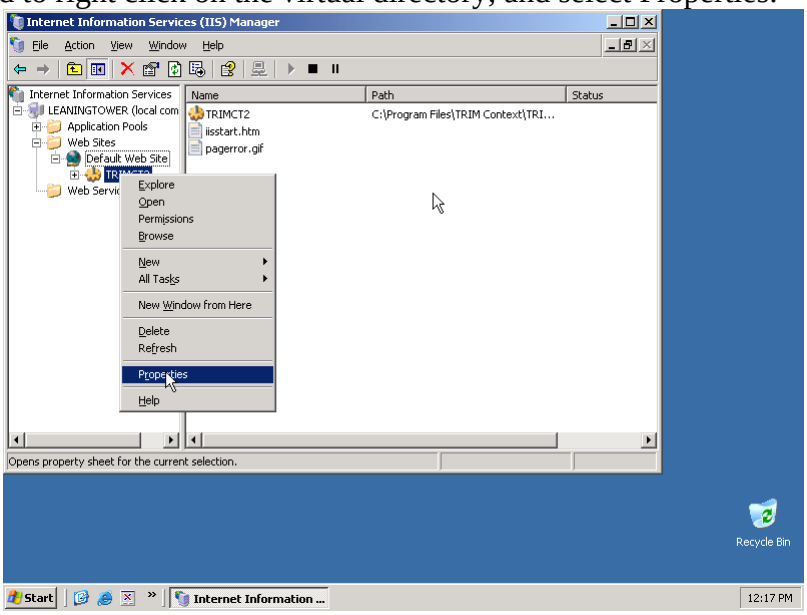
Specify some permissions, so they match the ones shown below:



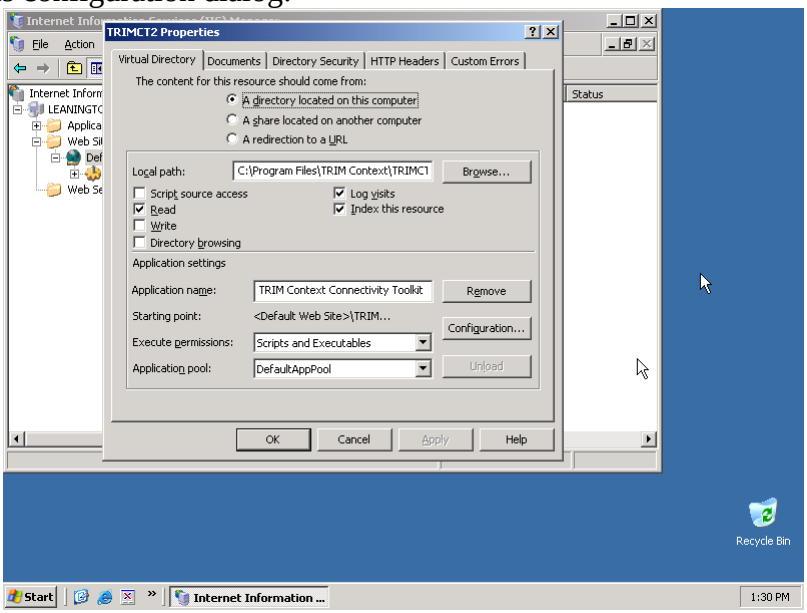
You've finished making the virtual directory:



You then need to right click on the virtual directory, and select Properties:

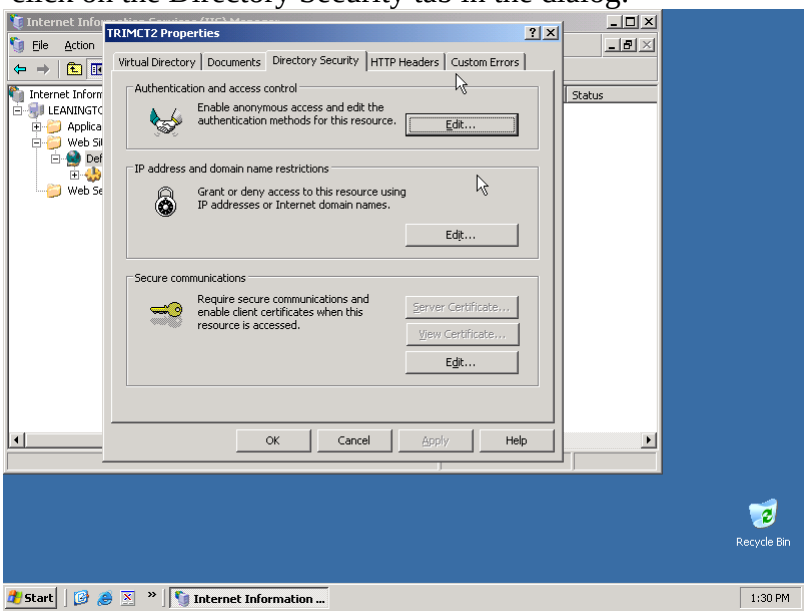


You'll see this configuration dialog:

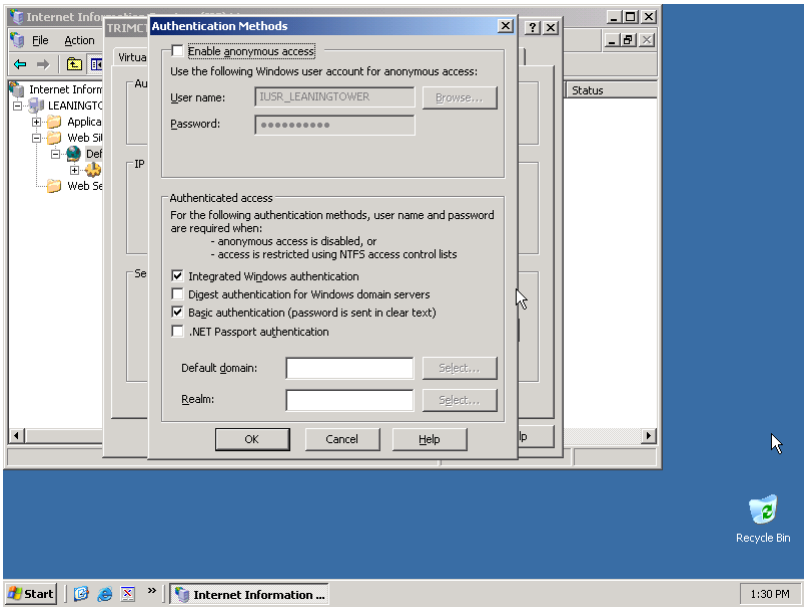


Pay special attention in this dialog to configuring the directory on disc to point to the location of the TRIM Connectivity Toolkit installation. If you are going to run more than one copy of the TRIM Connectivity Toolkit, then you'll need to copy this directory to another location, as the configuration file must be located in this directory as well.

Depending on your needs you might also want to set up a separate application pool for this instance of the TRIM Connectivity Toolkit (consult an IIS Administrator for advice on this). Now click on the Directory Security tab in the dialog:



Click on the edit button for the Authentication and Access Control group box. You'll see this dialog:



TOWER Software recommends that you not enable anonymous access to the TRIM Connectivity Toolkit, as this would leave the toolkit unable to determine which user was connecting to TRIM (effectively all connections will appear to TRIM to be coming from the user configured in this section if it is enabled).

TOWER Software recommends that you enable Integrated Windows Authentication and Basic Authentication.

For more information, contact an IIS administration specialist.

The Configuration File

The configuration for the TRIM Connectivity Toolkit is stored in a file named web.config which is located in the TRIM Connectivity Toolkit directory. There is a lot configuration stuff in this file which is related to the ASP.NET runtime environment and outside the scope of this document, but there are some values which are relevant to a deployment of the TRIM Connectivity Toolkit. Here's an example of the relevant section of the configuration file:

```
<appSettings>
        <add key="dbid" value="IT" />
        <add key="transferTempPath" value="%SYSTEMP%" />
        <add key="traceMode" value="true" />
        <add key="readOnly" value="false" />
        <add key="logName" value="" />
</appSettings>
```

The applications setting tag is where all of the configuration for the TRIM Connectivity Toolkit resides. The keys here have the following meanings:

Key	Meaning	Comments
dbid	The TRIM Dataset to connect to	
transferTempPath	The location to store temporary files in	The special %SYSTEMP% resolves to the Microsoft Windows temporary directory
traceMode	Turn on the active logging mode	This mode logs all events to occur in the toolkit, even if they're not errors. It helps TOWER Software support staff determine what is occurring in the toolkit for field support, but writes large amounts of information to the Windows Event Log.
logName	The name of the Windows Event Log to write to	An empty string means to use the default TRIM Connectivity Toolkit log source created by the MSI installer. Another common option is "Application", which writes to the Windows Application Event Log source.

readOnly	If true, no updates can occur	This will force an error to be returned if the caller attempts to update any information in TRIM via this instance of the TRIM Connectivity Toolkit

Conclusion

In this chapter we have discussed how to install the TRIM Connectivity Toolkit, as well as the various configuration options which are available.

3. Getting Started

Introduction

The first thing to do before beginning development against the TRIM Connectivity Toolkit is to make sure you can access “trim.asmx” file. This can be done by pointing any web browser at the host machine and supplying authentication details as needed. These details should match the LAN logon for the user configured in TRIM. Once authenticated, you should see a page like this:

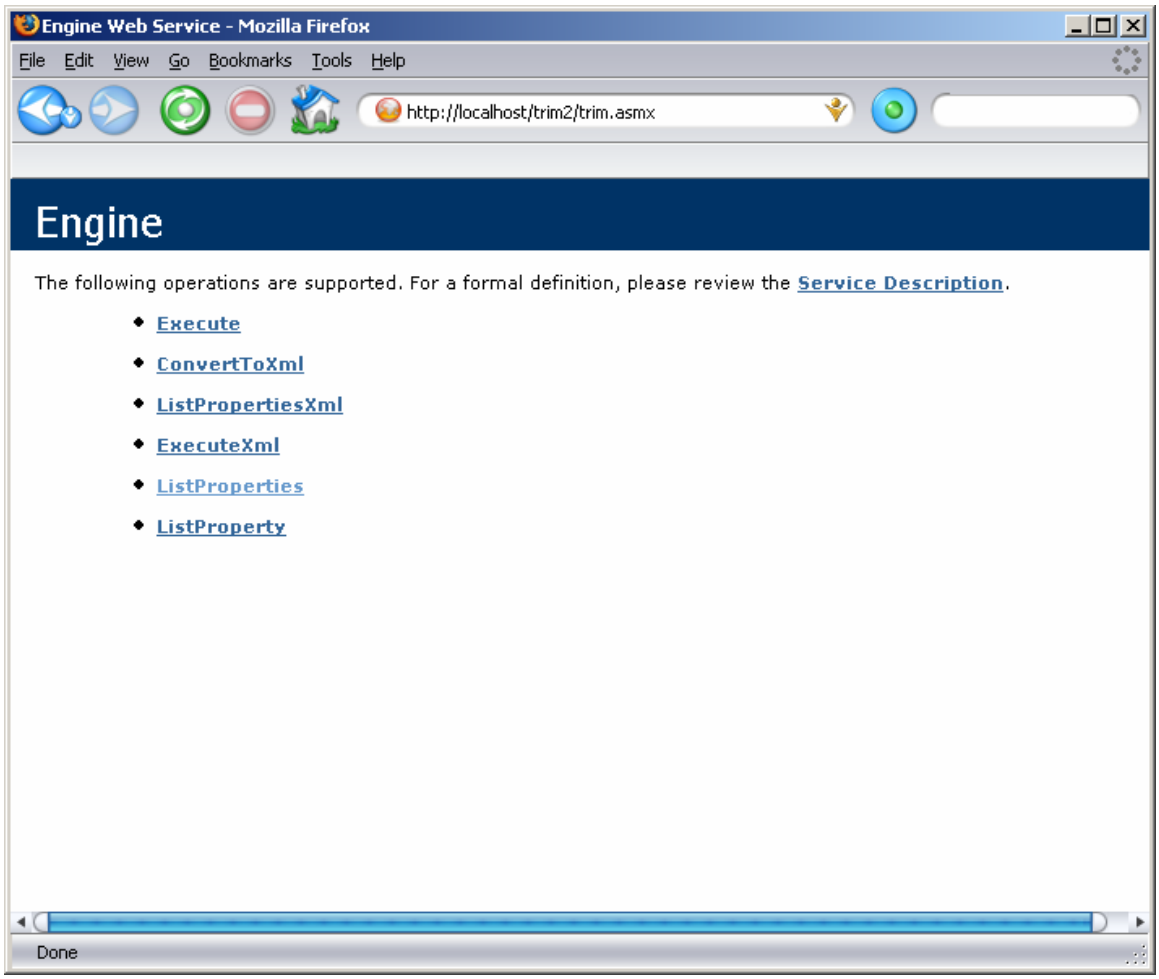


Figure 1 – TRIM Connectivity Toolkit as seen through a web browser

It is worth mentioning that you do not have to develop on the machine that is running the TRIM Connectivity Toolkit. The server machine may be on the other side of the room, across the world or exist only as a virtual machine.

A Note on IDEs

For this guide, we’re presuming that you’ll be programming in Microsoft’s Visual Studio .NET. To describe its objects, the TRIM Connectivity Toolkit uses the Web Services

Description Language (WSDL). Plenty of other development IDE's support WSDL and will assist the developer in writing code – but will interpret WSDL differently. This could mean that your client side objects may behave a little differently than described here. If you're having real trouble, contact the API support team at TOWER Software – apisupport@towersoft.com.au (see Chapter 15 on page 67).

To develop against the TRIM Connectivity Toolkit in Visual Studio, you'll need to install a few things:

- .NET runtime version 1.1
- Web Service Extensions 2.0 SP3 (this is installed by the TRIM Connectivity Toolkit installer)

If you aren't using Visual Studio, you'll need the equivalents for your chosen IDE.

Creating a Reference to the TRIM Connectivity Toolkit

To set a reference to the TRIM Connectivity Toolkit, select "Add Web Reference" from the project menu. The Add Web Reference dialog appears:

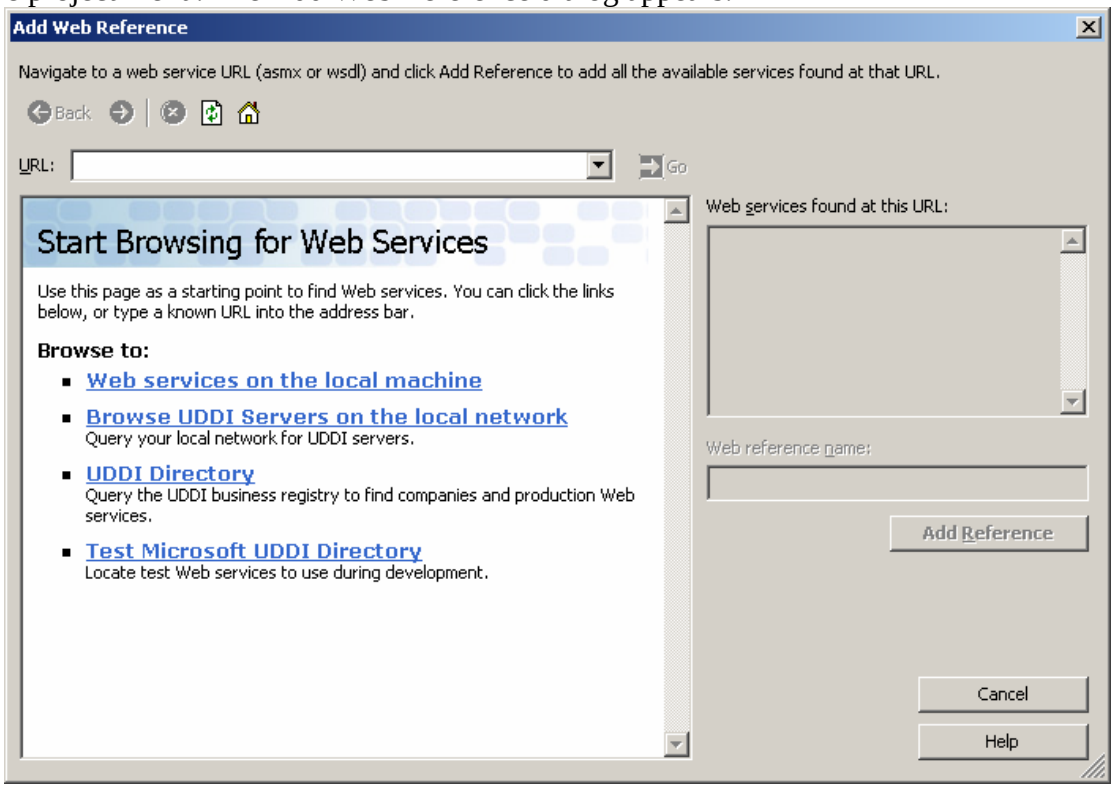


Figure 2 – Visual Studio's Add Web Reference Dialog

Type the URL of your configured Web Service into the url box – It's likely that you'll now get a logon dialog that prompts you for your username and password. Once you've been authenticated, the TRIM Connectivity Toolkit appears as follows:

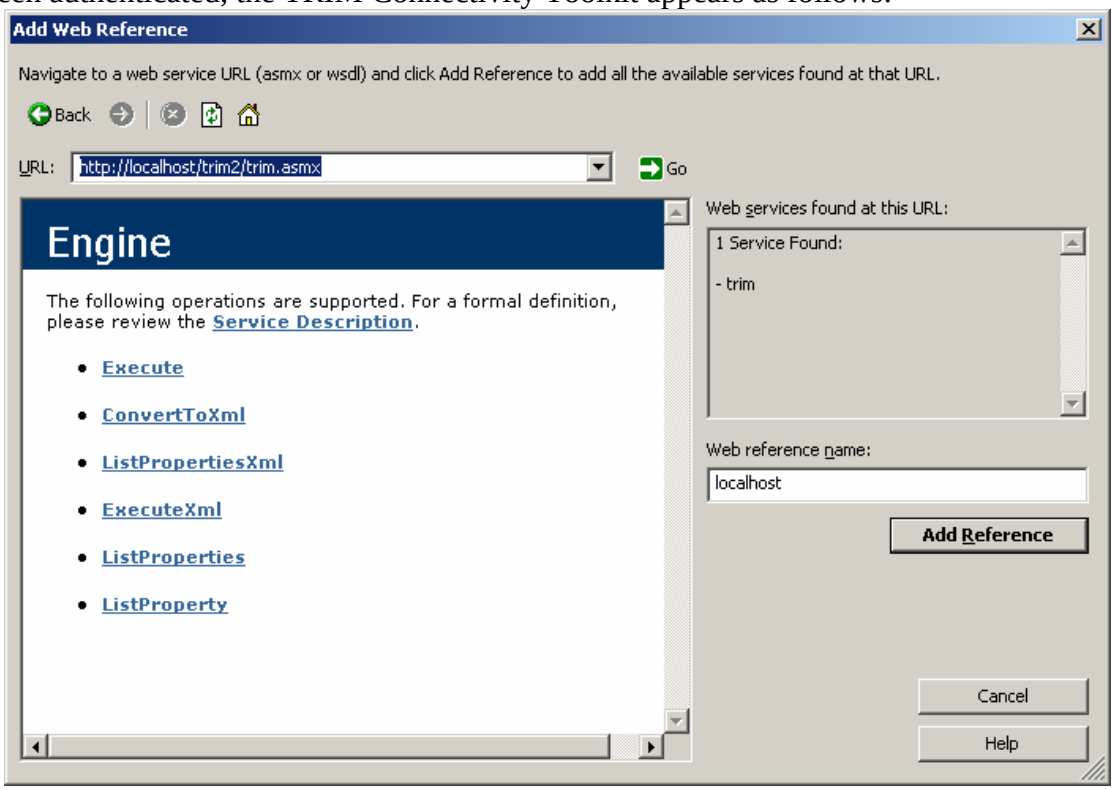


Figure 3 - the TRIM Connectivity Toolkit as seen through Visual Studio's eyes…

Note the word "Engine" there – that's the main object in the TRIM Connectivity Toolkit, which we'll discuss more soon. For now, you can click the "Add Reference" button to get Visual Studio to read the WSDL and create objects on your machine to help call the TRIM Connectivity Toolkit.

It is worth mentioning that developers should give the name of the web reference something more meaningful than "localhost" (as in figure 3).

4. Basic Searching

Introduction

Search is the most fundamental operation that any TRIM API can provide. Due to the request/response nature of the TRIM Connectivity Toolkit architecture, every call done to the Web Service is stateless. This means the TRIM Connectivity Toolkit will never remember what a client application was looking at previously. Almost any tasks done using the TRIM Connectivity Toolkit will, by necessity, require a search operation.

The RecordSearch Operation

The RecordSearch Operation exposes the TRIM Record Searching model. TRIM has a powerful and flexible model for searching for records, and in the TRIM Connectivity Toolkit, this is represented as a RecordSeach Operation. The RecordSearch Operation has no methods itself, just a collection of items that represent the search clauses.

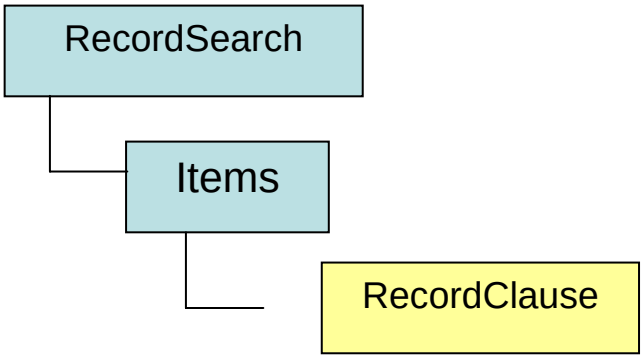


Figure 1 – The RecordSearch Operation. You can have as many RecordClauses as you want

In order to conduct a search, the process is to create a Record Search Operation, create one or more RecordSearch Clauses, and to append those clauses to the RecordSearch Object. Once that's been completed, append the search to the TrimRequest, and use the Engine object to execute that request.

About Clauses

If you view the object browser in your IDE you can see that the vast majority of objects created for the TRIM Connectivity Toolkit are RecordSearchClause Objects. These represent the different kinds of searches you can do, and they differ in the number and types of parameters that they take in order to perform a search. Most of these objects contain a property called "type" which allows you to further specify the kind of search you want to do. A typical example, one using text search:

```
RecordStringSearchClause clause = new RecordStringSearchClause();
clause.Type = RecordStringSearchClauseType.TitleWord;
clause.Arg = "reef";
```

The “type” of property on RecordStringSearchClause takes a value from an enumeration of RecordStringSearchClauseTypes. If we look at the enumeration in the object browser, we can see:

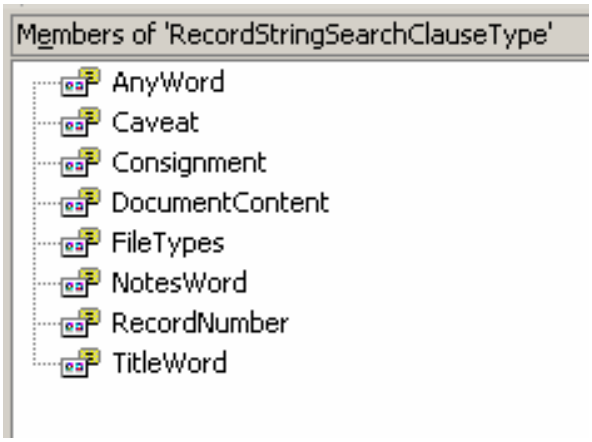


Figure 2 - The RecordStringSearchClauseType as seen in the Visual Studio Object Browser

Generally speaking, if a Clause has a type property, it will correspond to an enumeration which lists all the search clauses of that type. There are exceptions – for instance the RecordClassificationSearchClause doesn’t have a type. Instead, it has a ClassificationUri property where you pass in the Uri of the classification you’re looking for. This is because there is only one record clause which takes a classification Uri.

Here is an example of setting up a basic search operation. As with any TRIM Connectivity Toolkit operation, the search consists of a TrimRequest object, which has one or more operations. The request is then passed through to the engine object, which returns a TrimResponse.

Example 1 – A Title Word Search

```
// Construct a request object
TrimRequest request = new TrimRequest();

// Construct a RecordStringSearchClause, with type
// TitleWord, and argument "reef"
RecordStringSearchClause clause = new RecordStringSearchClause();
clause.Type = RecordStringSearchClauseType.TitleWord;
clause.Arg = "reef";

// Construct a record search, and put our search clause in it
RecordSearch search = new RecordSearch();
search.Items = new RecordClause[] { clause };
// If we had more than one clause, it would look like:
// search.Items = new RecordClause[] { clause1, clause2, clause3 }

// Put our search operation into our TrimRequest
request.Items = new Operation[] { search };

// Send it off. Whatever comes back will be in response
Engine engine = new Engine();
engine.Credentials = new System.Net.NetworkCredential(username, password);
TrimResponse response = engine.Execute(request);
```

In this example, the final line that executes the request would return two result objects – a SuccessResult, indicating that the search completed successfully, and an EndResult.
If an operation executes successfully, the Connectivity Toolkit will return you a SuccessResult. If the operation fails, the Connectivity Toolkit will return an ErrorResult with an error message. There will be a SuccessResult or ErrorResult for each operation you execute.

An EndResult indicates that the Connectivity Toolkit has dealt with everything, and has now forgotten about you. There will always be a single EndResult.

Getting Results

In the previous example, we didn't enquire about any of the properties of the Records, the TRIM Connectivity Toolkit conducted a search, noted the successful completion of the search, and finished. This may seem ultra-pedantic, but the reality is that TRIM Records can contain an enormous amount of data.

If you consider the relationship between Records, Locations and other Records, which are properties of a single Record, the possibility for TRIM to get locked in a recursive death-spiral if it returned everything it knew about a record in the results is quite high – so in order to alleviate this, the developers have left it entirely up to you. Additionally only returning the data that you need is the most effective way to use network resources.

The Fetch Operation

The Fetch Operation is the way of telling the TRIM Connectivity Toolkit what information to return to the client application. It's a comprehensive series of properties that can be used to obtain any information stored in TRIM about a record.

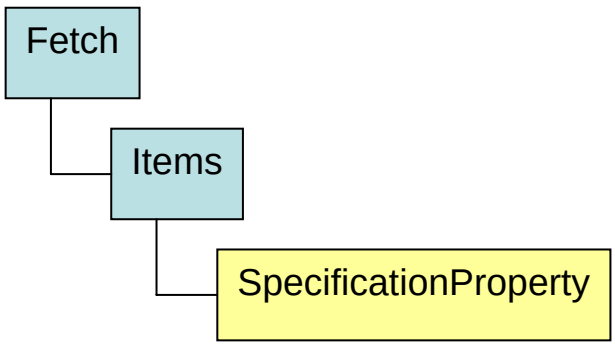


Figure 3 - The Fetch Operation. You can have as many SpecificationProperties as you want

If we look into the Fetch Operation, we can see that it's essentially a container for a collection of SpecificationProperty objects – for each piece of information you want to return from the Web Service, you need to supply a SpecificationProperty. The SpecificationProperty contains several properties that determine the manner in which

information is returned to you, (more about those later) but the probably the most important property to set when creating a SpecifcationProperty is the Name Property that uniquely identifies the value of the metadata you want to return. Note that a Fetch is only useful when combined in a TrimRequest with a search operation.

```
SpecificationProperty rectitleSpec = new SpecificationProperty();
rectitleSpec.Name = "rectitle";
```

This example tells the TRIM Connectivity Toolkit that we're looking for the title of the record to be present in the result set. To complete this, we need to append the SpecificationProperty to the Fetch operation:

```
Fetch fetch = new Fetch();
fetch.Items = new SpecificationProperty[] { rectitleSpec };
```

Combining these code snippets with the first example by including the SpecificationProperty and adding the Fetch operation to the TrimRequest, a sample that will return the title of each record found is produced.

Example 2 – A Search Returning Record Titles

```
// Construct a request object
TrimRequest request = new TrimRequest();

// Construct a RecordStringSearchClause, with type
// TitleWord, and argument "reef"
RecordStringSearchClause clause =
    new RecordStringSearchClause();
clause.Type = RecordStringSearchClauseType.TitleWord;
clause.Arg = "reef";

// Construct a record search, and put our search clause in it
RecordSearch search = new RecordSearch();
search.Items = new RecordClause[] { clause };

// This SpecificationProperty says we want the title
SpecificationProperty rectitleSpec = new SpecificationProperty();
rectitleSpec.Name = "rectitle";

// And this one says we want the date the record was Created
SpecificationProperty recdatecreatedSpec =
    new SpecificationProperty();
recdatecreatedSpec.Name = "recdatecreated";

// Create a new Fetch Operation, and add our SpecficationProperties
Fetch fetch = new Fetch();
fetch.Items = new SpecificationProperty[] { rectitleSpec };

// Put our search operation AND our fetch operation into the
// TrimRequest
request.Items = new Operation[] { search, fetch };

// Send it off. Whatever comes back will be in response
Engine engine = new Engine();
engine.Credentials = new System.Net.NetworkCredential(username, password);
TrimResponse response = engine.Execute(request);
```

Information on SpecificationProperties other than recTitle can be found at the end of this chapter.

Obtaining Results using the TrimResponse

The TrimResponse object contains a loosely federated collection of results– each one relating directly to an operation, and one that indicates the batch is complete. For example, if your request contains two operations, you'll get back three results – one for each operation, and one to mark that processing is complete.
The best way to process results is to iterate over the items returned, and inspect each one:

Example 3 – Iterating Through TrimResponse Items

```
    foreach (Result result in response.Items)
    {
        switch (result.GetType().Name.ToString())
        {
            case "EndResponse":
                // Code to handle the EndResponse
                textBox1.Text += "EndResponse!\r\n"
                break;

            case "SuccessResult":
                // Code to handle the SuccessResult
                textBox1.Text += "SuccessResult!\r\n"
                break;

            default:
                // Code to handle whatever else, etc…
                textBox1.Text += "Other Result!\r\n"
                break;
        }
    }
```

If you're preparing large and complex request, you can tag each operation with an ID, which will be returned against the corresponding result. The ID can be any string identifier you want. This is especially useful for asynchronous request, where it will tell you which request has finished

Example 4 –TrimResponse Items with IDs

```
RecordSearch search = new RecordSearch();
search.Id = "TheSearch";
search.Items = new RecordClause[] { clause };

// Additional code

foreach (Result result in response.Items)
{
    switch (result.GetType().Name.ToString())
    {
    case "SuccessResult":
            if (result.Id == "TheSearch")
            {
                textBox1.Text += "It was my search that worked!\r\n";
            }
            break;

        default:
            // Code to handle whatever else, etc…
            textBox1.Text += "Code to handle other stuff\r\n";
            break;
    }
}
```

There are several different kinds of results, each with different behaviors. For now, we'll only concentrate on the more commonly encountered results – SuccessResult, EndResult, FetchResult and ErrorResult.

The SuccessResult

For most operations that don't actively return information, the SuccessResult is the TRIM Connectivity Toolkit's way of saying everything was performed without any errors being returned. The SuccessResult has no other information about an operation that completed other than the ID that may have been assigned to the calling process. It's generally good practice to label all your operations with IDs, and to check them when receiving the response. This ensures that you can accurately handle results and operations.

The ErrorResult

If an operation failed, then an ErrorResult is returned in the place of a SuccessResult. The ErrorResult object contains properties that will give a unique error code and a string description of the error. The TRIM Connectivity Toolkit ability to handle batch operations means that the failure of one operation automatically results in the failure of any operation after the failed operation in the request.

Example 5 – Handling an ErrorResult

```
TrimResponse response = engine.Execute(request);
foreach (Result result in response.Items)
{
    switch (result.GetType().Name.ToString())
    {
        case "Error":
            ErrorResult err = (ErrorResult) result;
            if (result.Id == "TheSearch")
            {
                textBox1.Text += "Search operation Failed\r\n";
                textBox1.Text += "Error Code: " + err.ErrorNumber.ToString();
                textBox1.Text += "Error Message was " + err.Message;
            }
            break;
        default:
            // Code to handle whatever else, etc…
            break;
    }
}
```

FetchResult

The FetchResult contains all the information that's been requested about the results of a search. All of this data can be found in the Objects Collection – a collection of TrimObjects, which contain information about the object returned by the TRIM Connectivity Toolkit and contains a collection of objects that map directly to the SpecificationProperties passed in as part of a fetch operation. Additionally, the FetchResult contains the Count (the number of records returned by the search) and the ResultsTruncated properties. If your search exceeded the limit specified in the original

Fetch Specification, this Boolean will be set to true (and any result after the limit specified in the Fetch will be missing)

Let's have a look at the TrimObject objects that make up the FetchResult. We can see that it's made up of a collection of Value objects, which contain the data asked for with the SpecificationProperties. It also contains the URI for the object in question, and information on its version.

Each Value in the Values potentially also has another Value collection associated with it in the Children property. Iterating over the Values collection will return to us whatever properties our original SpecificationProperty objects attached to the original Fetch object.

EndResponse

The EndResponse is simply confirmation that the TRIM Connectivity Toolkit has finished processing. An EndResponse will always be returned.

ListProperties

Now you know how to search for records and how to specify what it is that you want to see in your results.

The nature of the SpecificationProperty means that you need to specify the name of the property, but as we've previously discussed, there could be easily a thousand properties available at any time.

Originally there was some discussion of perhaps shipping a book with the TRIM Connectivity Toolkit that listed all the properties, but seeing as that wasn't exactly practical, there is a method included on the engine object called ListProperties. To use it, you pass in the name of the object you're interested in, and it returns to you a list of all the properties available on the object.

Example 6 – List Properties

```
textBox1.Text = "LIST PROPERTY\r\n";
textBox1.Text += "Building Engine\r\n";
Engine engine = new Engine();
engine.Credentials = new System.Net.NetworkCredential(username, password);

PropertyDescription[] descs = engine.ListProperties("classification");

foreach (PropertyDescription desc in descs)
{
    textBox1.Text += desc.Name + " (" + desc.Caption + "): \r\n";
}
```

Note that this method is provided solely as a reference material for developers. Providing a list of all the SpecificationProperties that exist for a Record is of very limited use for an end user.

Other SpecificationProperty Properties

Besides the name property, the SpecificationProperty object includes two additional properties, both of which deal with the fact that properties on TRIM objects can be objects themselves. These two properties are the Children Collection and the ForView Property. ForView relates to a visible version of the string, instead of a parsable one (eg 32000 vs 32mbs).

Conclusion

A search done through the Connectivity Toolkit will be made up of two things. The Search object will contain the information needed to find the object or objects in TRIM. The Fetch object will contain one or more SpecificationProperties objects, each of which indicates a property of the object (or objects) to be returned.

After executing a search, the TrimResponse object that is returned will contain one or more Result objects. A SuccessResult indicates that the operation executed without errors. An ErrorResult contains the details of why an operation failed. A FetchResult is made up of the values matching the SpecificationProperties passed in through the Fetch object. Finally, an EndResult indicates the last Result object.

5. Primary Operations

Introduction

There are several “generic” operations in the TRIM Connectivity Toolkit – operations that are used the same way across different objects. Searching is covered in depth in its own chapters (Chapter 4 on page 23 and Chapter 10 on page 53).

The operations covered in this chapter include preparing an object to be modified, creating a new object and deleting an existing object.

Is For Update

The IsForUpdate property on searches is a way of specifying if the results of a search are to be modified or not. It was implemented because (as you will see later in this chapter), it is trivial to accidentally modify or delete a large number of objects in one go.

Example 1 – Setting IsForUpdate

```
RecordStringSearchClause rss = new RecordStringSearchClause();
rss.Arg = "Reef";
rss.Type = RecordStringSearchClauseType.TitleWord;

RecordSearch search = new RecordSearch();
search.Items = new RecordClause[] {rss};
search.IsForUpdate = true;
```

Whenever an object is to be altered, the IsForUpdate property on the search that locates the object must be set to True. If it is set to False (as it is by default), any following update or delete operation will fail.

SpecificationProperty Children

Some properties on an object are objects themselves. For example, the author of a Record is actually a Location object. A SpecificationProperty that is asked to return an object will return the Uri of that object which can then be used in other searches or in an injector (see Chapter 11, page 58).

Sometimes the properties of the returned object will need to be accessed directly. That’s what the idea of SpecificationProperty children is for. Each SpecificationProperty is capable of taking a collection of SpecificationProperties. This collection is applied to the object found by the parent property.

Example 2 – Using SpecificationProperty

```
// Get the Location we want to look at
SpecificationProperty rectitleSpec = new SpecificationProperty();
rectitleSpec.Name = "recAuthorLoc";

// Get the property of the Location
SpecificationProperty locname = new SpecificationProperty();
locname.Name = "locSurname";
```

```
rectitleSpec.Children = new PropertyBase[] {locname};

// Create a new Fetch Operation, and add our
// SpecficationProperties
Fetch fetch = new Fetch();
fetch.Items = new SpecificationProperty[] { rectitleSpec};

// Put our search operation AND our fetch operation into the
// TrimRequest
TrimRequest req = new TrimRequest();
req.Items = new Operation[] { search,fetch };

TrimResponse resp = engine.Execute(req);
```

The above example will return the URI of the author Location, along with the surname of the Location. It should be noted that SpecificationProperty children can be nested – so it would be possible to find the manager of the author in the example, and the manager's organization, and so on.

InputProperties

InputProperties are virtually identical to SpecificationProperties, but go the other way. While you use a SpecificationProperties to find data on an object, you use InputProperties to manipulate or add that data. InputProperties have the same names (and so can be found the same way) as SpecificationProperties – using ListProperties.

Example 5 – Creating an InputProperty

```
InputProperty title = new InputProperty();
title.Name = "recTitle";
title.Val = "New Record";
```

An InputProperty has a Name, which uses a SpecificationProperty name (so recTitle, locSurname, rtyName, etc), and a Val, which is the value you are suggesting for that property.

Modifying Objects

Updating properties is the simplest way to modify the metadata of a Record. You simply assign a new value of the correct data type to the named property of the object. Field-level verification is carried out, and an error will be returned if the modification you attempted violated some of the TRIM business rules.

Modifying Metadata with the Update Operation

The simplest way to update data in a TRIM record is to modify the named properties on the object. Obviously, this can only be done on properties that are not marked as read-only. This includes most of the Date properties, certain Location properties (AuthorLoc, AddresseLoc and OtherLoc) and miscellaneous properties such as External Id, Priority, Accession Number and Foreign Barcode for the Record object. You can if a property is read only when you use the ListProperties method on the Engine object.

Assuming the property can be updated, it is the Update object that does the work. An Update object takes in an array of InputProperties which is will apply to the object or objects found in the search immediately preceding it. These InputProperties are made up of two important values. The first is the name of the property you are modifying – so "recTitle" or "locSurname". The second is the value of the new property – so "Updated Timesheet" or "Smith".

Example 3 – Updating Metadata

```
Engine engine = new Engine();
engine.Credentials = new System.Net.NetworkCredential(username, password);
TrimRequest request = new TrimRequest();

RecordUrisSearchClause clause = new RecordUrisSearchClause();
clause.Uris = new string[] { "313" };
RecordSearch search = new RecordSearch();
search.IsForUpdate = true;
search.Items = new RecordClause[] { clause };

InputProperty rectitleInput = new InputProperty();
rectitleInput.Name = "rectitle";
rectitleInput.Val = "The New Title Is This!";

Update update = new Update();
update.Items = new InputProperty[] {rectitleInput};

request.Items = new Operation[] { search, update };

TrimResponse response = engine.Execute(request);

foreach (Result result in response.Items)
{
        switch(result.GetType().Name.ToString())
        {
                case "FetchResult":
                        FetchResult fetchResult = (FetchResult) result;
                        foreach (TrimObject obj in fetchResult.Objects)
                        {
                                foreach (Value property in obj.Values)
                                {
                                        txtOutput += "Got " + property.Name + ":";
                                        txtOutput += property.Val;
                                }
                        }
                        break;
                }
        }
}
```

Will it save? Verifying Records

There are two operations to assist in making sure changes are valid. These are the Verity and VerifyCreate operations. These work like verifying a Record prior to saving it in the COM SDK.

If an attempt is made to update a Record that fails, the error returned by the TRIM Connectivity Toolkit will give details on why the operation failed – for example a value for a required field wasn't specified on a Create operation.

User Defined Fields

TRIM Context allows a large number of user-defined fields to be assigned to Records and Locations. Due to the loosely bound approach to updating properties taken by the TRIM Connectivity Toolkit, the act of updating User Defined Fields is precisely the same as for updating a native TRIM metadata field – create an InputProperty, assign it to the UDF and give it the required value. If the attempt to update the field fails (for example the business rules limit the field to a particular series of lookup values, and the value supplied is outside of those) the TRIM Connectivity Toolkit will return an error result for the update.

All UDFs are described in SpecificationProperties and InputProperties with the prefix "udf:", followed by the UDF's title. For example, a UDF entitled "Current License", the code would look like

Example 4 – Specifying a User Defined Field

```
SpecificationProperty driver = new SpecificationProperty();
driver.Name = "udf:Current License";
```

A list of UDFs is returned along with the other SpecificationProperties when the ListProperties method is executed on an object.

Create

Being able to create new objects is of vital importance to most TRIM setups – after all, what's the point of records management if you can't manage new documents? During the development stage of the original TRIM Connectivity Toolkit, there was a Create object for each possible object. As each object requires a different minimum set of parameters to be created, this seemed like a good approach. However, it proved too unwieldy to implement and did not scale well to future development. Instead, a single Create object has been implemented to cater for all objects. This Create operation is really just a special type of Update operation.

It is worth noting that since you are dealing with a new object, not one found through a search, there is no IsForUpdate property to set.

Create.Items

The Create object takes an array of InputProperties. As mentioned previously, each object has a subset of properties that *must* be set in order for the Create operation to be successful. This array of InputProperties can set every property on the object – but it *must* set that minimum subset.

Example 6 – Creating a Location

```
//Build input properties
InputProperty lastname = new InputProperty();
lastname.Name = "locsurname";
lastname.Val = "Able";
```

```
//Build the CreateLocation object
Create create = new Create();
create.TrimObjectType = "location";
create.Items = new InputProperty[] {lastname};

//Build the request
request.Items = new Operation[] {create};
```

Errors

If the minimum subset for the object is not specified, then the operation will fail and an ErrorResult will be returned. The ErrorResult will specify the property it failed on, but will only indicate one property. If multiple properties are missing, it will only report on the first it encountered.

Example 7 – Failed Create Error Message

> Could not save changes to (empty) with identifier 0: Please enter a name for this Location. (TRIM Connectivity Toolkit unique id = 1000237)

Unfortunately, there isn't any way of knowing for certain what properties are needed to successfully create an object. Referring to the SDK documentation may be of assistance, but you may need to resort to trial and error.

Delete

Using a Delete object is fairly simple – create the object and pass it along after a search. It is important to remember that a delete is very powerful; the TRIM Connectivity Toolkit bypasses many of the checks done in the normal TRIM interface. It is assumed that the developer will implement similar safeguards to help protect users.

Example 8 – Deleting a Record

```
ShortcutRecordUri sru = new ShortcutRecordUri();
sru.Uri = "553";
sru.IsForUpdate = true;

Delete delete = new Delete();

TrimRequest request = new TrimRequest();
request.Items = new Operation[] {sru, delete};
```

As mentioned previously, the Delete operation was the main reason for the implementation of the IsForUpdate property. A Delete operation will not be successful if IsForUpdate on the associated Search object is not set to true.

Dates

The way that dates are handled in the TRIM Connectivity Toolkit has changed for the version released with TRIM Context 6. Dates are now formatted using the sdWebService string type from the TRIM Context COM SDK, which produces strings in this format:

```
Sun, 06 Nov 1994 08:49:37 GMT
```

The TRIM Connectivity Toolkit will always return the time in GMT. This complies with RFC 822 (used for SMTP and HTTP). You need to use this format when you pass strings into the TRIM Connectivity Toolkit as well. Most importantly, remember to include time zone information.

Conclusion

By the end of this chapter you should have a good over view of the generic operations that can be performed by the TRIM Connectivity Toolkit. These operations include creating and deleting an object, how errors and dates are handled, and how to access User Defined Fields.

6. Child Lists

Introduction

There are some objects within TRIM which can contain references to other objects. An example is that there can be many locations associated with a record object. Most SDK programmers will be familiar with the recAuthorLoc property for the author location for the record. This chapter introduces the mechanism which underlies this shortcut property, and documents a much more powerful method of accessing these associated objects.

Child Objects

In the example above, the locations associated with a record are called child locations of that record. Each record has a list of child objects, and the locations will be referred to by objects of type recLocation in that list. This is done because it is entirely possible that a single record will have more than one author. To extract all of the authors of a record, an SDK application would iterate through the child list, looking for objects of type recLocation which are authors.

Using the TRIM Connectivity Toolkit to access these child list items is essentially the same as accessing any other property. The biggest differences are that it is possible to have more than one property value returned for a single property request (for example, in the more than one author case), and that you need to use conditional schematics to ensure that you only update one of these child list items at a time.

Let's look at some examples.

Fetching Children

This code fetches a number of properties of a location, including a child list item:

Example 1 – Fetching Children

```
trimct.Engine engine = new trimct.Engine();
engine.Credentials = System.Net.CredentialCache.DefaultCredentials;

LocationUrisSelect lls = new LocationUrisSelect();
lls.Uris = new string[] {"1"};
lls.Limit = 1;

trimct.Fetch fth = new Fetch();
SpecificationProperty rts = new SpecificationProperty();
rts.Name = "locSurname";
SpecificationProperty rts2 = new SpecificationProperty();
rts2.Name = "locFullFormattedName";

SpecificationProperty rts3 = new SpecificationProperty();
rts3.Name = "locEmailAddress";
SpecificationProperty rts4 = new SpecificationProperty();
rts4.Name = "locGivenNames";
SpecificationProperty rts5 = new SpecificationProperty();
rts5.Name = "locGender";
SpecificationProperty rts6 = new SpecificationProperty();
```

```
        rts6.Name = "locLocType";
        SpecificationProperty rts7 = new SpecificationProperty();
        rts7.Name = "child:locEAddress";

        fth.Items = new SpecificationProperty[] {rts, rts2, rts3, rts4, rts5, rts6, rts7};

        trimct.TrimRequest req = new trimct.TrimRequest();
        req.HideVersionNumbers = true;
        req.Items = new trimct.Operation[] {lls, fth};

        textBox2.Text = engine.ConvertToXml(req);
        // Execute the request
```

The SpecificationProperty with the name property set to “child:locEAddress” is a request for a child list item. All such requests will start with a “child:”, in much the same way that requests for User Defined Fields will start with “udf:”.

The return results will vary depending on how many locEAddresses exist in the child list for this location. Here is an example (in XML form):

Example 2 – XML Response to Example 1

```
<?xml version="1.0"?>
<TrimResponse xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SearchResult>
    <FoundCount>1</FoundCount>
  </SearchResult>
  <FetchResult ResultsTruncated="false" Count="1" TrimObjectType="location">
    <Objects>
      <TrimObject Uri="1" Version="2005-06-28T00:44:35+10:00">
        <Values>
          <Value Name="locSurname" Val="Skywalker" ErrorOccurred="false">
            <Children />
          </Value>
          <Value Name="locFullFormattedName" Val="Luke Skywalker" ErrorOccurred="false">
            <Children />
          </Value>
          <Value Name="locEmailAddress" Val="luke @gmail.com" ErrorOccurred="false">
            <Children />
          </Value>
          <Value Name="locGivenNames" Val="" ErrorOccurred="false">
            <Children />
          </Value>
          <Value Name="locGender" Val="0" ErrorOccurred="false">
            <Children />
          </Value>
          <Value Name="locLocType" Val="4" ErrorOccurred="false">
            <Children />
          </Value>
          <Value Name="child:locEAddress" Val="501" ErrorOccurred="false">
            <Children />
          </Value>
          <Value Name="child:locEAddress" Val="504" ErrorOccurred="false">
            <Children />
          </Value>
        </Values>
      </TrimObject>
    </Objects>
  </FetchResult>
  <EndResponse />
</TrimResponse>
```

The child list item being returned is the one with the name “child:locEAddress”. If there were more than one object to return, then the name will be repeated in the list of return results.

You can of course specify the sub-properties of a child list item, just as you can for any other property which is an object. Remember that the child list item isn’t the destination object type though, it has an intermediate type. For example, a recLocation isn’t a location, but it does point to a location (with other information as to the type of the linkage included as well). For more information on accessing sub-properties, refer to the “Basic Properties” chapter earlier in this manual.

Changing Just One Child List Item

Now that we’ve found a child list item, let’s see how to change its properties. The change itself happens just like it does in the chapter entitled “basic properties”, but in this case we need to specify which of the children to change otherwise they will all be changed (this conditional behaviour didn’t exist in the first release of the TRIM Connectivity Toolkit for TRIM Context 5.2). This is done by including an additional property to the InputProperty specifying the new value – the TargetChildUri. This value should be set to the Uri of the child object to change.

Here’s an example of how to do this:

Example 3 – Changing a Child List Item

```
InputProperty childLocationInput = new InputProperty();
childLocationInput.Name = "rlcLocation";
childLocationInput.Val = "14";

InputProperty childRecLocationInput = new InputProperty();
childRecLocationInput.Name = "child:recLocation";
childRecLocationInput.Children = new PropertyBase[] {

childLocationInput };
childRecLocationInput.TargetChildUri = "221";

Update upd = new Update();
upd.Items = new InputProperty[] { childRecLocationInput };

req = new TrimRequest();
req.Items = new Operation[] { sru, upd } ;

// Execute the request
```

This will have the result of only changing the child list item of type recLocation, with the uri of 221.

Creating a Child List Item

The TRIM Connectivity Toolkit can also create child list items. This is done with the CreateChildItem operation, which takes a type for the child object, and a list of the input properties to set on the new object. The way this operation is used is to create or

otherwise select the objects you want to create the child list item for, and then execute the CreateChildItem operation. Here is an example:

Example 4 – Creating a Child List Item

```
CreateChildItem cci = new CreateChildItem();
cci.ChildObjectType = "reclocation";

trimct.InputProperty loc = new trimct.InputProperty();
loc.Name = "rlcLocation";
loc.Val = "inject:katie";

trimct.InputProperty type = new trimct.InputProperty();
type.Name = "rlcRecLocType";
type.Val = "Contact";

cci.Items = new InputProperty[] { loc, type };

// Execute the request
```

In this example we are creating a new contact on a record. The new contact is the location which was previously found using Uri injection. More coverage of Uri injection can be found in the “Injectors” chapter elsewhere in this documentation.

Conclusion

In this chapter we have discussed the purpose of child lists on TRIM objects, how to find child list items by treating them as properties, how to change the value of a child list item by using a conditional change, and finally how to create a new child list item.

7. Record Operations

Introduction

This chapter will cover operations related to the Record object in TRIM. It will not talk about any search objects. While there are many Record-specific searches in the TRIM Connectivity Toolkit, these are covered in the Record Search chapter (XX).

AddRendition

The AddRendition operation is similar to the CheckIn object, except it is used to add additional documents to a record. Before it can be used successfully, the new rendition document needs to be sent to the Connectivity Toolkit machine (using the Upload method). Like CheckIn, AddRenditions needs the UploadId that is returned from the Upload operation (refer to CheckIn and CheckOut, page 48, for more information on Uploading documents), and has a number of RenditionTypes that can be specified

Example 1 – Adding a Rendition

```
//Perform an Upload, and then remember the UploadId
ShortcutRecordUri sru = new ShortcutRecordUri();
sru.Uri = "559";
sru.IsForUpdate = true;

AddRendition ar = new AddRendition();
ar.RenditionType = RenditionTypes.Redaction;
ar.Description = "This is the redacted version";
ar.UploadId = uplid;

req.Items = new Operation[] { sru, ar };
resp = engine.Execute(req);
```

AppendNotes

While a record's notes can be modified using the "recNotes" specification property, the AppendNotes operation provides slightly different functionality.The idea is you don't need to know the current value of the notes, you're just appending. The "AddUserStamp" property adds the user's name and the time before the notes are appended.

Example 2 – Appending Notes

```
ShortcutRecordUri sru = new ShortcutRecordUri();
sru.Uri = "559";
sru.IsForUpdate = true;

AppendNotes an = new AppendNotes();
an.AdditionalNotes = "I've made those corrections";
an.AddUserStamp = true;

TrimRequest req = new TrimRequest();
req.Items = new Operation[] {sru, an};
```

AttachKeyword

This allows a Thesaurus term to be applied to a record. The KeywordUri is the Uri of the Thesaurus term to be applied to the record.

Example 3 – Attaching a Keyword

```
ShortcutRecordUri sru = new ShortcutRecordUri();
sru.Uri = "559";
sru.IsForUpdate = true;

AttachKeyword ak = new AttachKeyword();
ak.KeywordUri = "6";

TrimRequest req = new TrimRequest();
req.Items = new Operation[] {sru, ak};
```

CheckInMailMessage

This object expands the normal CheckIn operation, allowing additional email-related information to be entered. The various recipients of the email can be specified through the Recipients collection, and whether the email should be left checked out or if it's a new revision.

You can also handle any attachments associated with the mail message. These need to be uploaded prior to the CheckInMailMessage operation (as does the mail message its self – see CheckIn and CheckOut, page 48). Any attachments need to be handed to the CheckInMailMessage operation as MailAttachment objects.

Example 4 – Checking In Mail Message

```
MailAttachment ma = new MailAttachment();
ma.DisplayName = "comparisons.jpg";
ma.DocumentExtension = "jpg";
ma.UploadId = attachmentUplid;

MailRecipient mr = new MailRecipient();
mr.MailAddress = "simon.dugard@towersoft.com.au";
mr.DisplayName = "Simon Dugard";
mr.Type = MailRecipientTypes.To;

CheckInMailMessage cim = new CheckInMailMessage();
cim.UploadId = emailUplid;
cim.Attachments = new MailAttachment[] {ma};
cim.Recipients = new MailRecipient[] {mr};
cim.SentDate = "27/9/05";

ShortcutRecordUri sru = new ShortcutRecordUri();
sru.Uri = "563";
sru.IsForUpdate = true;
```

CompleteCurrentAction

Including this operation in a Request will complete the current action on the associated record.

Example 5 – Completing a Current Action

```
ShortcutRecordUri sru = new ShortcutRecordUri();
sru.Uri = "561";
sru.IsForUpdate = true;

CompleteCurrentAction cca = new CompleteCurrentAction();

TrimRequest req = new TrimRequest();
req.Items = new Operation[] {sru, cca};
```

CreateCopy

CreateCopy duplicates an existing record. You can specify the type of copy you're making, and make a suggestion for a new number.

Example 6 – Creating a Copy of an Existing Record

```
ShortcutRecordUri sru = new ShortcutRecordUri();
sru.Uri = "559";
sru.IsForUpdate = true;

CreateCopy cc = new CreateCopy();
cc.Type = CopyType.Part;
cc.SuggestedNumber = "19/05-02";

TrimRequest req = new TrimRequest();
req.Items = new Operation[] {sru, cc};

Engine engine = new Engine();
engine.Credentials = System.Net.CredentialCache.DefaultCredentials;

TrimResponse resp = engine.Execute(req);
```

DeleteRendition

If you need to remove a rendition, the DeleteRendition object is the thing to use. It is used in conjunction with a search operation, much like the AddRendition operation mentioned earlier in this chapter. The Uri of the Rendition needs to be specified as well.

Example 7 – Deleting a Rendition

```
ShortcutRecordUri sru = new ShortcutRecordUri();
sru.Uri = "561";
sru.IsForUpdate = true;

DeleteRendition dr = new DeleteRendition();
dr.ChildUri = "502";

TrimRequest req = new TrimRequest();
req.Items = new Operation[] {sru, dr};
TrimResponse resp = engine.Execute(req);
```

Finalize

The Finalize operation is used to finalize a record. It is similar to the CompleteCurrentAction operation in that it can be passed across without any parameters being set. It can be considered simply a flag that can be set.

Example 8 – Finalizing a Record

```
ShortcutRecordUri sru = new ShortcutRecordUri();
sru.Uri = "561";
sru.IsForUpdate = true;

Finalize fin = new Finalize();

TrimRequest req = new TrimRequest();
req.Items = new Operation[] {sru, fin};
TrimResponse resp = engine.Execute(req);
```

Conclusion

In this chapter we have covered the Record-specific operations that aren't covered in other parts of this documentation (specifically the Searching chapters). The operations covered included adding and removing a rendition, completing a current action and finalizing a record.

8. Non-Record Operations

Introduction

There are several operations which can be performed on objects within TRIM which aren't Records via the TRIM Connectivity Toolkit. This chapter covers the set of those operations which isn't covered in other chapters of this manual. The operations are broken into a series of groups, each of which is discussed in turn.

Access Control

Access Control Lists (ACLs) can also be manipulated with the TRIM Connectivity Toolkit. For further information on how ACLs work in TRIM, refer to the TRIM SDK documentation.

HasAccess

The HasAccess operation is used to test if a given user has access to a given object. The operation takes the Uri of the location to check for access for (a location often being a person within TRIM), and the type of ACL to check for. For more information on how the various ACL list types in TRIM interact, refer to the TRIM SDK documentation for the HasAccessBase().

The HasAccessResult returns a text description of the type of access which was checked for in a string suitable for display to your users. It also contains a list of access control items describing the access.

The HasAccess operation is performed on the objects returned from the previous search.

RemoveAccess

To remove access to objects you would use the RemoveAccess operation. This operation takes the same arguments as the HasAccess operation – an ACL type, and the Uri of the location to operate on.

AppendAccess

To append a location to an ACL, use the AppendAccess operation. This operation takes the ACL type, and a list of location Uris to append to the ACL. It then performs the ACL manipulation without you needing to know what other items are in the ACL. This was done this way because if you grabbed the ACL, modified it, and then handed the new ACL to the TRIM Connectivity Toolkit that would require at least two requests, and it's possible that someone else would have changed the ACL while you were processing it before your second request. This would have resulted in their changes to the ACL being overwritten by yours.

This operation returns a SuccessResult.

Deleting an Object

The TRIM Connectivity Toolkit allows you to delete one object at a time with the Delete operation. This operation operates on the result of the last search operation. As such, care should be taken not to accidentally delete a large number of objects.

Information about the TRIM dataset and current connection

ConnectionInfo

The ConnectionInfo operation returns information about the TRIM dataset in use and the currently connected user. The operation doesn't take any arguments, and returns a ConnectionInfoResult result. This result contains the following values:

Value returned	Description
CurrentUserUri	The Uri of the currently connected user
CurrentUserNickName	The NickName of the currently connected user
CurrentUserLogin	The network login for the current user
LicenseName	The name of the license holder for this TRIM dataset
LicenseNumber	The license number for this installation

IsLicensed

Similarly, the IsLicensed operation is used to test if a given piece of TRIM functionality is licensed by the current TRIM installation. This can be used by client applications to determine if functionality within their applications should be enabled or disabled, or if in fact that given application will work in a given TRIM environment. The IsLicensed operation takes a LicenseTypes enumeration entry as an argument, and returns a Boolean wrapped in an IsLicensedResult if that functionality is enabled for the current TRIM dataset.

The LicenseTypes enumeration contains these entries:

```
Workflow = 0,
RecordsManagement = 1,
SpaceManagement = 2,
EnterpriseCachedStore = 3,
DocumentContent = 4,
WebServer = 5,
GuestGateway = 6,
DocumentManagement = 7,
ClassifiedSecurity = 8,
```

```
CustomResourcesAvailable = 9,
AnnotateRedact = 10,
DocumentAssembly = 11,
MeetingManager = 12
```

User Labels

UserLabels are a user specific grouping paradigm implemented by the *ice* product in Context 6. For more information on the mechanics of UserLabels, refer to the TRIM SDK documentation.

ApplyUserLabel

To apply a UserLabel to a set of objects, you use the ApplyUserLabel operation. This operation takes a UserLabel Uri, and applies this UserLabel to all of the objects returned by the previous search operation.

RemoveFromUserLabel

Similarly, to remove a UserLabel, use the RemoveFromUserLabel operation. This operation takes the Uri of a UserLabel, and removes that label from all of the objects returned by the previous search.

Conclusion

In this chapter we have covered the operations not covered by other chapters of the documentation. In several instances we have recommended that you refer to the TRIM SDK documentation – in any case because the TRIM Connectivity Toolkit is built on top of the TRIM SDK this is a good idea anyway.

9. CheckIn and CheckOut

Introduction

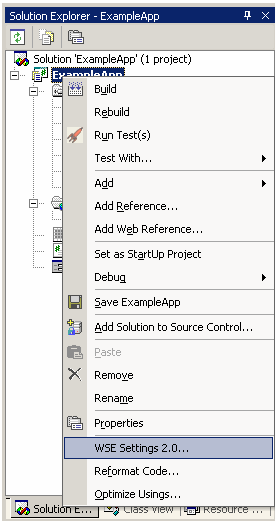
One of the great things about TRIM is that it lets you store various electronic documents. So a Record may contain your current invoice list as a Word document or photos from the last company picnic. Documents can be checked out, modified and then checked back in while TRIM keeps track of the changes.
Naturally, this ability has been included in the TRIM Connectivity Toolkit, allowing a user on the other side of the world to download a document, modify it and then check it back in.

WSE2

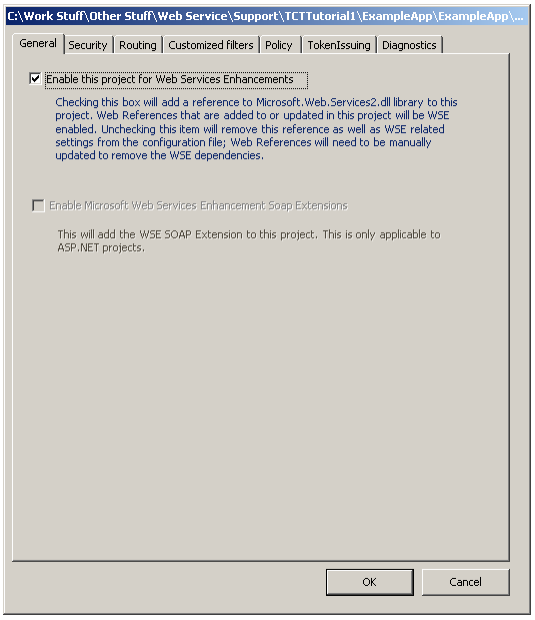
Web Service Enhancements, or WSE is a vital part of the file transfer and document manipulation objects in the TRIM Connectivity Toolkit Web Service if you're using a Microsoft programming language to call the Web Service. It is needed to implement WS-Attachments. Other language vendors will have equivalent products. The Inline transfer method exists for developers who don't want to or cannot use WSE. Before any coding can be started or any example run, you'll have to make sure you have WSE2 Service Pack 3 (or equivalent) properly installed and setup.

WSE2 SP3 can be downloaded from Microsoft at http://www.microsoft.com/downloads/. If you install WSE2 on a machine that you'll be developing TRIM Connectivity Toolkit applications on with Microsoft Visual Studio, you'll need to select the Visual Studio Developer option.

Once installed, you'll need to restart Visual Studio. The WSE2 SP3 will now be integrated with Visual Studio, and can be included in projects. Once you've opened (or created) the solution you want to use WSE with, go to the Solution Explorer and right click on the solution. Near the bottom of the menu that appears is "WSE Settings 2.0". Select this, and a new dialog will appear.



Under the General tab is a check box marked “Enable this project for Web Services Enhancements”. This box needs to be checked. Once that’s done, click OK and update the web reference.



You’ll need to include the lines “using Microsoft.Web.Services2” and (depending on what you’re using) “using Microsoft.Web.Services2.Attachments” and “using Microsoft.Web.Services2.Dime”.

If the solution fails to recognize the WSE-related objects (i.e. EngineWSE), you may need to update the WebReference after turning on Web Service Enhancements.

EngineWSE

Operations like Download and Upload using the Attachment transfer type require a different Engine object to what we have used previously – the EngineWSE object. Unlike the Engine object, the EngineWse is not actually part of the TRIM Connectivity Toolkit, but instead is part of the WSE2 package.

If you're using the WSE version of the Web Reference, then it is best practice to always use that version of the Engine, and never use the non-WSE version of the Engine.

Downloading

The first operation we need to look at is the Download. Download is used to retrieve a specific document from the TRIM database and transfer it to the client's computer. The first step is to create the EngineWSE object instead of the normal Engine.

Once the engine has been established, it's time to create a search of some kind to find the record with the attachment you want to download. For this example, we'll just assume we already know the Uri of the record we want. Once the Download object has been created, a few parameters need to be set. Checkout is where you decide if you want to check the document out or get a local copy. After the request has been built, handed over to the server and the request has been returned, you'll need to loop through the results as normal, this time paying particular attention for the DownloadResult.

Example 1 – Downloading via Attachment

```
// Create an EngineWse object
EngineWse exec = new EngineWse();
exec.Credentials = System.Net.CredentialCache.DefaultCredentials;

// Assume we know the URI of the record
ShortcutRecordUri rus = new ShortcutRecordUri();
rus.Uri = "12";

// Create the Download object
Download down = new Download();
down.Checkout = false;
down.Comments = "TRIM Connectivity Toolkit Example";
down.MaximumTransferBytes = 0;
down.TransferInset = 0;
down.TransferType = TransferTypeType.attachment;
```

Example 2 – Downloading via Inline

```
// Create an EngineWse object
EngineWse exec = new EngineWse();
exec.Credentials = System.Net.CredentialCache.DefaultCredentials;

// Assume we know the URI of the record
ShortcutRecordUri rus = new ShortcutRecordUri();
rus.Uri = "12";

// Create the Download object
Download down = new Download();
down.Checkout = false;
down.Comments = "TRIM Connectivity Toolkit Example";
down.MaximumTransferBytes = 0;
down.TransferInset = 0;
```

```
                down.TransferType = TransferTypeType.inline;

                req.Items = new Operation[] { rus, down };
                TrimResponse resp = sdk.Execute(req);
                byte[] data;

                foreach(Result res in resp.Items)
                {
                        if(res.GetType().Name.ToString() == "DownloadResult")
                        {
                                DownloadResult dres = (DownloadResult) res;
                                data = Convert.FromBase64String(dres.Base64Payload);
                                using(FileStream output = new FileStream(@"pic.jpg",
FileMode.Append))
                                {
                                        output.Write(data, 0, data.Length);
                                        output.Flush();
                                        output.Close();
                                }
                        }
                }
```

Uploading

Once you've finished with the document you just downloaded – or if you have a new document you want to put into TRIM – you'll need the Upload operation. Like Download, Upload uses the EngineWSE object if you're using the Attachment transfer type.

Upload takes an electronic document you specify and places it on the web server machine. You then attach the document to a record in TRIM, which moves it to the e-store. So obviously you need a Record in TRIM to attach to attach the document to. If there isn't a record for the document already in TRIM, you'll have to create it first. Otherwise, you'll just need to find it, but remember to mark the Record for update.

To grab the electronic document, you'll need to use a SoapConext object and an Attachment object. You give the path of the document you're uploading to the attachment object, and then assign the attachment object to the SoapContext object.

The Upload object needs to have an upload ID and a transfer type. The Upload ID links the Attachment object with the Upload object – it's basically a way to keep track of where the file ends up.

Once all the objects have been set up correctly, the Upload and the Search objects need to be passed into a request and fired off across the web. If the process is successful, you will get returned an Upload Result with another upload ID.

Example 2 - Uploading

```
                SoapContext reqc = engine.RequestSoapContext;
                Attachment att = new Attachment("Incoming", "application/octet-stream",
"../../image.jpg");
                reqc.Attachments.Add(att);

                Upload upl = new Upload();
                upl.AttachmentId = "Incoming";
                upl.TransferType = TransferTypeType.attachment;
```

```
TrimRequest req = new TrimRequest();
req.Items = new Operation[] { sru, upl };
TrimResponse resp = engine.Execute(req);
string uplid = "foobar";

foreach(Result res in resp.Items)
{
        switch(res.GetType().Name.ToString())
        {
                case "UploadResult":
                        UploadResult uplr = (UploadResult) res;
                        uplid = uplr.UploadId;
                        Console.WriteLine("Upload id is: " + uplid);
                        break;
        }
}
```

Check In

The Upload example from the previous section has placed the file on the server, but the document still needs to be checked into TRIM. That's where CheckIn comes in. The CheckIn operation will take the document (it finds it using the Upload ID returned when you Upload the document), attaches it to a record and moves the electronic copy to the e-store.

Example 2 – Check In

```
CheckIn cin = new CheckIn();
cin.Comments = "What a nice puppy";
cin.DocumentExtension = ".jpg";
cin.FailOnWarning = true;
cin.KeepCheckedOut = false;
cin.UploadId = uplid;

req.Items = new Operation[] { sru, cin };
resp = engine.Execute(req);
```

Conclusion

Manipulating documents through the TRIM Connectivity Toolkit requires Microsoft's Web Server Enhancements (service pack 2) or equivalent. Some users have encountered a problem within Visual Studio .NET where the WSE objects are not recognized, but this can be corrected by updating the solution's Web Reference.

Uploading and checking in a document needs to do done in two separate executes, as the Upload ID is needed for the CheckIn operation, but cannot be passed directly from the Upload operation.

10. Advanced Searching

Introduction

In this chapter we are going to explore some of the more complex search options available to the TRIM Connectivity Toolkit developer. Record Location searches are useful when trying to locate Records by Locations – such as the author or the current location.

This chapter will also cover building searches over several properties. These Boolean searches could, for example, find all the records registered between two dates and which have the title word “reef” in them.

Record Location Searches

Searching for records by their related Locations is extremely useful for tracking the life of a Record – who first created it, who has modified it, and where the record currently is. To address this functionality, the TRIM Connectivity Toolkit has the RecordLocationTypeSearchClause.

Through the LocationType property, you can use the RecordLocationTypeSearchClause to find out the current assignee of a record, the actual owner of the location or who has the record currently checked out.

Example 1 – Searching Records by Locations

```
RecordLocationTypeSearchClause ass = new RecordLocationTypeSearchClause();
ass.LocationUri = 17; //Peter Abbot
ass.Type = RecordLocationTypeSearchClauseType.Location;
ass.LocationType = SearchLocationTypes.Current;
```

Boolean Searching

In earlier chapters we looked at how to search a TRIM dataset for a specific criterion. Boolean searching is the term given to searching over more than one criterion, and it’s one of the more powerful features of the Connectivity Toolkit.

Logical Clauses

Special clauses exist to combine two or more search clauses. Collectively, these are called the Logical Clauses, since each of them represents a logical separator. Logical separators are used to combine two items. One of the really nice things about logical separators is that you can combine two or more statements together to create some quite complex algorithms. There are two types of logical separators available in the TRIM Connectivity Toolkit – AND clauses and OR clauses.

For the following examples, imagine we have three records in our TRIM dataset - “Great Barrier Reef” registered in 1/1/2000, “Reefs of the World” registered on 1/1/1990 and “Special Coral” registered on 1/1/2001.

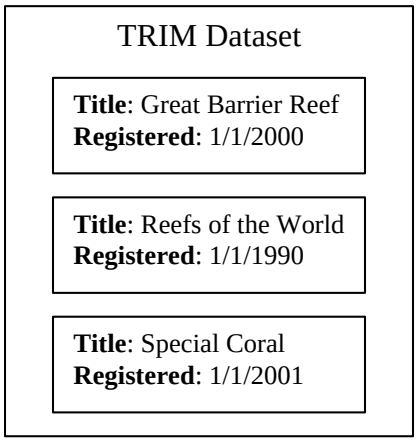


Figure 1 – Sample Records in a TRIM Dataset

AND Clause

An AND clause means that an object must meet both of the search clauses to be included in the found set. An example of an AND search would be looking for records with a TitleWord of "reef" registered between today and the 1st April 1990.

Based on Figure 1, we would only get "Great Barrier Reef" returned.

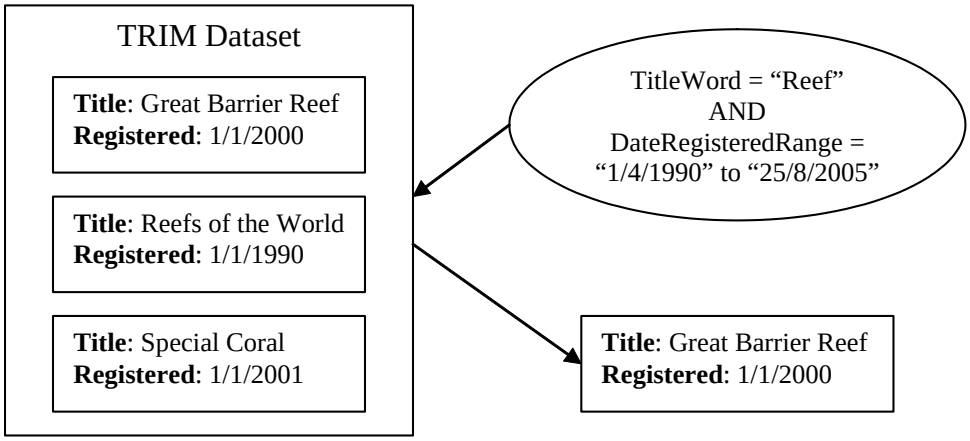


Figure 2 – AND Search Results

OR Clause

An OR clause is used to look for objects than meet either clause. It's worth noting that an OR clause will also return item that meets both criteria – it's basically a less picky version of the AND clause. An example of an OR clause would be looking for all the Records with the TitleWord of "great" or "coral". If you were using an OR clause, based on our example above, you would receive two results.

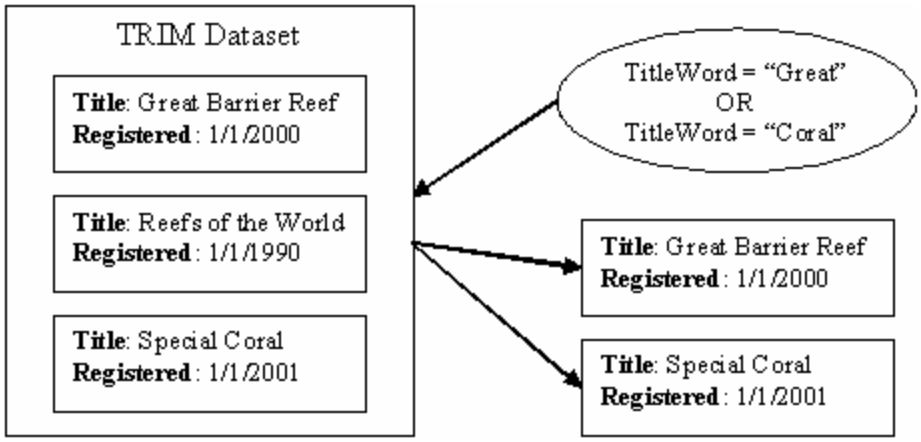


Figure 3 – OR Search Results

If the same search was done with an AND clause, neither of the Records would have been found. Only a Record with "Great" *and* "Coral" would be returned.

NOT Clauses

There is a third type of logical clause we haven't talked about yet. Unlike AND and OR, the NOT clause is not a Boolean clause. Instead, think of a NOT clause as a modifier. A NOT clause will return the opposite result to what would normally be returned. So while a search for records with a TitleWord of "reef" on our sample dataset would return 2 of 3 records, the same search with a NOT clause would return the only 1 record.

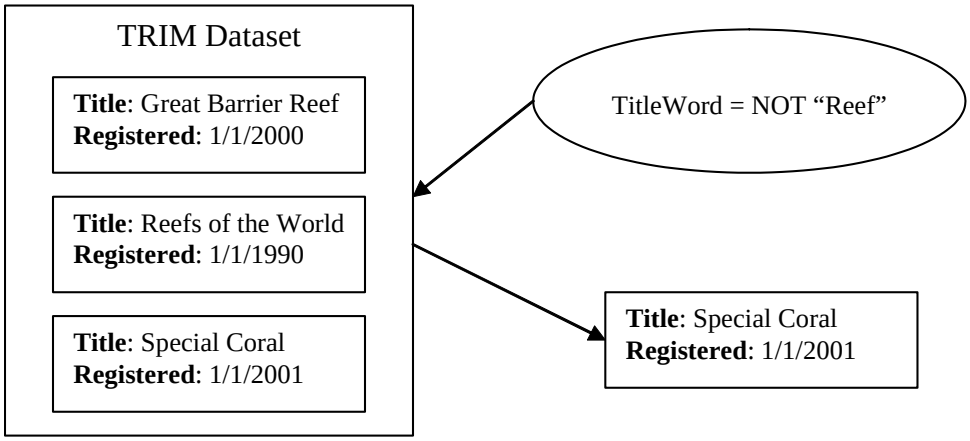


Figure 4 – NOT Search Results

Boolean Search Format

Boolean searches in the Connectivity Toolkit use the reverse Polish notation. A normal equation has the two parameters separated by an operation. So adding together two numbers would look like:

2 + 27

Reverse Polish works differently. Instead of parameter-operation-parameter, the reverse Polish format is operation-operation-parameter. So the example above would be:

2 27 +

So when applied to searches in the TRIM Connectivity Toolkit, the notation looks something like:

SearchClause1 SearchClause2 LogicalClause

Building a Boolean Search

You can combine any search clauses with a Boolean separator to produce a more complex search. So, as we talked about earlier, you can search by the RecordTitle and DateRegistered, or two different Surnames or any other combinations of SpecificationProperties.

By this stage you should be able to construct two different searches, send off both of these searches and get back two separate datasets. It's a fairly simple matter to combine the two searches.

As you may have realized, the AND, OR and NOT clauses are individual clauses in the TRIM Connectivity Toolkit. They are created in the same way as other Clause objects, except they have no parameters that need to be set.

Example 2 – Creating an ANDed Search

```
RecordStringSearchClause titleword = new RecordStringSearchClause();
titleword.Type = RecordStringSearchClauseType.TitleWord;
titleword.Arg = "Reef";

RecordDateRangeSearchClause date = new RecordDateRangeSearchClause();
date.Type = RecordDateRangeSearchClauseType.DateRegistered;
date.StartTime = "1/4/1990";
date.EndTime = "25/1/2005";

RecordAndSearchClause and = new RecordAndSearchClause();

search.Items = new RecordClause[] {titleword, date, and};
request.Items = new Operation[] {search, fetch};
```

Multiple Boolean Conditions

So far we've talked about how to use the logical search clauses to search across two parameters. But it is possible to construct even more complex search queries by using multiple Boolean clauses.

Example 3 – Multiple Boolean Clauses

```
{titleword, date1, and, titleword, date2, and, or};
```

Dealing with multiple Boolean clauses can become very confusing very quickly. It may be easier to think of the parameters in pairs. So the example above would read as:

```
((titleword, date1, and), (titleword, date2, and), or)
```

If we translate the Polish notation into a more natural notation, we get:

```
((titleword and date1) or (titleword and date2))
```

Conclusion

While constructing Boolean searches can quickly become confusing if care is not taken, they are a powerful part of the TRIM Connectivity Toolkit. They allow you to either refine or expand your search conditions to cater for a much larger group of potential searches than would be possible with individual searches.

Record Location Searches allow Records to be searched on by their associated Locations. This allows electronic documents to be tracked as they pass between different individuals or to help track down the current location of physical Records.

11. Injection

Introduction

There are potentially many situations where a developer using the TRIM Connectivity Toolkit will need to perform some operations on an object but will not know until runtime which specific object it is. Based on previous chapters, the only option would be to perform two executes on the Engine – one to identify the specific object, and a second to carry out the operations.

To cater for this situation, the concept of "injection" was developed. While there are several different types of Injection, the basic workings are the same – rather than pass a Uri into a parameter, you can pass a string value that the TRIM Connectivity Toolkit can resolve for a Uri. Injection works anywhere you would use a Uri.

Uri Injection

Uri Injection is used in the situations where the Uri of the object is being found with some type of search, and then passed through to the next operation. Take, as an example, dealing with a RecordType Uri when creating a new Record. It is possible to do a search for the RecordType, have its Uri returned and then placed in a local variable before being passed into the Create object and sent off in a second execute.

Using Uri Injection means that these two executes can be grouped together – the search is carried out, and then the TRIM Connectivity Toolkit on the server side directly passes the result to the Create object using a FetchInjectionUri object.

The FetchInjectionUri object works a bit differently to a normal Fetch object. Instead of specifying what properties you want returned, you instead specify the FetchInjectionUri's ID. This ID can be any string, but it needs to be unique for the Execute. The FetchInjectionUri object then caches the Uri found by the search for later use in the request.

Dealing with Injected Uris

By now you should be familiar with at least some of the objects in the TRIM Connectivity Toolkit that take a Uri as a parameter - for example specifying the RecordType when creating a new Record.

For Uri injection to work, you first need a successful search. We'll assume for this example that you at least know the name of the RecordType you need. So you'll need a search that can find the Record Type. As mentioned earlier, you can set the ID of the injection object to any string, but it needs to be unique for the request.

Example 1 – Creating a Record using Inject

```
Engine engine = new Engine();
```

```
engine.Credentials = System.Net.CredentialCache.DefaultCredentials;

RecordTypeStringSelect rts = new RecordTypeStringSelect();
rts.Arg = "Document";
rts.Id = "recordtype";

InputProperty name = new InputProperty();
name.Name = "recTitle";
name.Val = "New Record";

InputProperty type = new InputProperty();
type.Name = "recRecordType";
type.Val = "inject:recordtype";

Create create = new Create();
create.Items = new InputProperty[] {type, name};
create.TrimObjectType = "record";

FetchInjectionUri fiu = new FetchInjectionUri();
fiu.Id = "recordtype";

TrimRequest req = new TrimRequest();
req.Items = new Operation[] {rts, fiu, create};
TrimResponse resp = engine.Execute(req);
```

The first InputProperty is pretty standard – it's just specifying the record title. The second InputProperty is what we're really interested in. Notice that instead of specifying the Uri, we've instead set it to "inject:recordtype". This is basically saying that this value will be set by the FetchInjectUri object.

As you've probably guessed, the reason the ID of the FetchInjectionUri object needs to be unique within a single request is that otherwise the TRIM Connectivity Toolkit doesn't know which Uri to inject where. In fact if you reuse a FetchId, the old value will be silently overwritten with the new one.

Things to Remember

Uri Injection replaces the need to search for an object and return its Uri before using the Uri in another request. Naturally, this is only useful when dealing with a single Uri. The operation will fail if the FetchInjectionUri has to deal with anything but a single Uri (so the search returns either nothing or more than one Uri). This prevents anything messy from happening if, for example creating a new record, your search returned two Record Types.

Name

If you know the name of the object, you can use the "name:" injector. The "name:" injector saves the need to do a shortcut name or similar search. When the TRIM Connectivity Toolkit detects the "name:" syntax, it carries out a relevant name-based search automatically, by constructing the object using its name as you would in the TRIM SDK.

Example 2 – Creating a Record using Name

```
Engine engine = new Engine();
engine.Credentials = System.Net.CredentialCache.DefaultCredentials;
```

```
InputProperty name = new InputProperty();
name.Name = "recTitle";
name.Val = "New Record";

InputProperty type = new InputProperty();
type.Name = "recRecordType";
type.Val = "name:document";

Create create = new Create();
create.Items = new InputProperty[] {type, name};
create.TrimObjectType = "record";

TrimRequest req = new TrimRequest();
req.Items = new Operation[] {create};
```

Conclusion

Injectors are designed to help minimize executes and make life easier for the developer. There are many situations where using an injector will not be useful – or even possible. For example, a common mistake is trying to use an injector to pass an upload ID (see CheckIn and CheckOut, page 48) to a CheckIn object. This will not work, as the upload ID does not resolve to a Uri.

12. Shortcut Operations

Introduction

To make development easier, several "shortcut" objects have been included in the TRIM Connectivity Toolkit. These shortcuts are designed to make some of the more common tasks within TRIM easier to perform, saving on the need to construct complex clauses. This means they are ideally suited to be used in conjunction with updates, injections, deletes and similar operations.

Records

As Records are really the key objects in the TRIM Connectivity Toolkit, it's natural that the lion's share of the Shortcut objects are related to Records.

ShortcutRecordNumber

Every Record in TRIM has a notionally unique Record Number within a given RecordType. This follows a pattern defined by the Record Type and can be manually entered by the user or set to be automatically generated by TRIM. Although the commonly used term is 'number', it is more correctly an identifier, as it is a string that may contain alphanumeric characters. To access this string – pass a SpecificationProperty with the name "recNumber" in with your fetch operation.

Example 1 - ShortcutRecordNumber

```
// Construct a request object
TrimRequest request = new TrimRequest();

ShortcutRecordNumber recNum = new ShortcutRecordNumber();
recNum.Id = "recordNumber";
recNum.RecordNumber = "2005/0059";

// Put our shortcut operation into our TrimRequest
request.Items = new Operation[] { recNum, fetch };
```

Note: TRIM stores the Record Number in two formats, expanded (e.g. "2002/0059") and compressed (e.g. "02/59"). Both can be passed to the *Number* property.

ShortcutRecordTitle

If you know the title of a record, then the Shortcut search that will be of most use will be ShortcutRecordTitle search. This shortcut search works like the title word search in the TRIM UI – it will return matches based on one or more words, and you can incorporate wildcards.

Example 2 - ShortcutTitleNumber

```
// Construct a request object
TrimRequest request = new TrimRequest();
```

```
ShortcutRecordTitle recTitle = new ShortcutRecordTitle();
recTitle.Id = "recordTitle";
recTitle.TitleWord = "reef";

// Put our shortcut operation into our TrimRequest
request.Items = new Operation[] { recTitle, fetch };
```

ShortcutRecordUri

The Unique Row Identifier or Uri of a record is an internal unique number that is transparent to the everyday user of TRIM. It is the primary key on the TSRECORD table in the database and provides an internal unique identifier for every record.
To instantiate a record by its Uri, you can use the ShortcutRecordUri Operation. It works in much the same way as ShortcutRecordNumber.

Example 3 - ShortcutRecordUri

```
// Construct a request object
TrimRequest request = new TrimRequest();

ShortcutRecordUri recUri = new ShortcutRecordUri();
recUri.Uri = "785";

// Put our shortcut operation into our TrimRequest
request.Items = new Operation[] { recUri, fetch };
```

Once an instantiated Record object has been returned by the either of these shortcut methods, the developer can access whatever properties were returned from the object. Reading this data requires inspecting the FetchResult, as we've done in Chapter 4 - updating these requires a new operation that we haven't yet explored – the Update operation.

ShortcutRecordUris

If you need to select multiple records, rather than use multiple instances of the ShortcutRecordUri object, there uses a ShortcutRecordUris operation. All of the records found by this search will have the associated operations performed on them – for example a ShortcutRecordUris search that is accompanied by an operation to change the record author to "Joe Blogs" will alter all of the matching records.

Example 4 - ShortcutRecordUris

```
// Construct a request object
TrimRequest request = new TrimRequest();

ShortcutRecordUris srus = new ShortcutRecordUris();
srus.Uris = new string[] {785, 786, 789};

// Put our shortcut operation into our TrimRequest
request.Items = new Operation[] { srus, fetch };
```

ShortcutRecordUris accepts an array of Uris, and so it is worth nothing that it can accept as few as one Uri.

ShortcutRecordDateRegistered

The final shortcut method for finding records searches the TRIM dataset for Records that were registered between a certain date range.

Example 5 - ShortcutDateRange

```
// Construct a request object
TrimRequest request = new TrimRequest();

ShortcutRecordDateRegistered srdr = new ShortcutRecordDateRegistered();
srdr.StartTime = "1/1/1990";
srdr.EndTime = "1/1/2005";

// Put our shortcut operation into our TrimRequest
request.Items = new Operation[] { srdr, fetch };
```

Locations

Currently there is only one Shortcut object for Locations.

ShortcutLocationName

ShortcutLocationName is a useful search for getting Locations with a specific name. It is worth noting that when search on “people” Locations, the Locations are searched by their surnames – in fact, the SpecificationProperty that is searched on is named “locSurname”.

Example 6 - ShortcutLocationName

```
ShortcutLocationName sln = new ShortcutLocationName();
sln.Name = "smith";

TrimRequest request = new TrimRequest();
request.Items = new Operation[] {sln, fetch};
```

Conclusion

Shortcut operations are designed to save time for the developer by simplifying some of the more common search operations. Combined with Uri Injection (see Chapter 11, page 58), Shortcuts are particularly powerful – allowing a developer to design considerably more efficient code.

13. XML Methods

Introduction

It's possible that some users of the TRIM Connectivity Toolkit will be unable to make use of the full SOAP object functionality of the TRIM Connectivity Toolkit. For example, you may have dumped XML data from a database and would rather use XSLT to put it in the Connectivity Toolkit request.

ExecuteXml() and ConvertToXml()

The TRIM Connectivity Toolkit supports these users by allowing for an XML string version of the TrimRequest object to be passed to a special method named ExecuteXml(). The string taken by ExecuteXml() is the XML form of the TrimRequest that was otherwise passed, and can in fact be generated by calling the ConvertToXml() method, which takes in a TrimRequest, and returns an XML string.

For example, let's take a simple TrimRequest and convert it to XML. Here is some code to setup a simple search:

Example 1 – Converting to XML

```
TrimRequest req = new TrimRequest();
req.HideVersionNumbers = true;
RecordSearch recSearch = new RecordSearch();

// Now we need to make one or more search clauses for that search. We only make
// one here, as we're only going to do a title word search.
RecordStringSearchClause titleWordClause = new RecordStringSearchClause();
titleWordClause.Arg = "protective";
titleWordClause.Type = RecordStringSearchClauseType.TitleWord;

recSearch.Items = new RecordClause[] { titleWordClause };
req.Items = new Operation[] { recSearch };

Engine sdk = new Engine();
sdk.Credentials = System.Net.CredentialCache.DefaultCredentials;
m_SimpleXmlData = sdk.ConvertToXml(req);
```

Which gives us this XML:

Example 2 – XML Result

```
<?xml version="1.0"?>
<TrimRequest xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <RecordSearch>
    <IsForUpdate>false</IsForUpdate>
    <Limit>0</Limit>
    <Sort1>None</Sort1>
    <Sort1Descending>false</Sort1Descending>
    <Sort2>None</Sort2>
    <Sort2Descending>false</Sort2Descending>
    <Sort3>None</Sort3>
    <Sort3Descending>false</Sort3Descending>
    <FilterFinalizedState>Both</FilterFinalizedState>
    <RecordStringSearchClause>
      <Arg>protective</Arg>
```

```
      <Type>TitleWord</Type>
    </RecordStringSearchClause>
  </RecordSearch>
  <HideVersionNumbers>true</HideVersionNumbers>
  <ProvideTimingResults>false</ProvideTimingResults>
  <ForceRealTimeCacheUpdate>false</ForceRealTimeCacheUpdate>
</TrimRequest>
```

You can use this XML as the basis for a more complicated XML string if it's needed. You can then execute this XML string like this:

Example 3 – Executing XML

```
Engine sdk = new Engine();
sdk.Credentials = System.Net.CredentialCache.DefaultCredentials;
sdk.ExecuteXml(xmlString);
```

Which would be the same as executing the original TrimRequest with Execute(). You are returned the results in XML form, in this case using my test dataset I get:

Example 3 – XML Result

```
<?xml version="1.0"?>
<TrimResponse xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SearchResult>
    <FoundCount>2</FoundCount>
  </SearchResult>
  <EndResponse />
</TrimResponse>
```

Because there is no fetch, nothing more is returned.

Conclusion

In this chapter we have demonstrated how to use the XML versions of the TRIM Connectivity Toolkit web methods.

14. Debugging and Error Logs

Introduction

The role of this chapter is to cover what to do when things don't work out perfectly with the TRIM Connectivity Toolkit. We provide some ideas about where to start with debugging your applications, how to determine if an error has occurred within the TRIM Connectivity Toolkit, and our preferred method of reporting errors to the API Support team (who are discussed in the "API Support" chapter later in this document).

Debugging

Debugging your use of proprietary software can sometimes be hard. TOWER has however put a lot of effort into automated regression and unit testing for the TRIM Connectivity Toolkit, so hopefully we haven't shipped you code which doesn't work. It is possible that you have encountered a bug in the TRIM Connectivity Toolkit however, and we discuss how to go about reporting those bugs if you find one.

The first step to debugging a problem with the TRIM Connectivity Toolkit is to step through your client code and see if you can determine what the error being returned by the TRIM Connectivity Toolkit is (if there is one). Often these error messages will say helpful things like "you have to specify a record type to create a record". If you receive and error message with such a hint, try implementing the hint and seeing if that helps.

If you need help interpreting the hint, or a less helpful error message is returned, then contact the API Support team as detailed in the "API Support" chapter for assistance with interpreting the message. All messages returned by the TRIM Connectivity Toolkit have a unique id associated with them, because it is possible that your error message was returned in a language that the TOWER development team doesn't speak. If this is the case, then the unique id gives them the English version of the string, as well as telling them exactly where in our code that message was created.

Logging

Another useful technique to debugging what is happening in the TRIM Connectivity Toolkit is to turn on the trace logging facility, as discussed in the "Installation" chapter. This will then log to the configured log in the Windows Event Log applet messages as to the progress of your request. These logs can help you determine what is happening with the request, as well as providing useful information to the API Support team.

Conclusion

In this chapter we've discussed some introductory debugging techniques for the TRIM Connectivity Toolkit. Contact the API Support team if you need more assistance.

15. Contacting TOWER API Support

API Support

TOWER Software has a team dedicated to providing developers with assistance with the TRIM SDK. Recently their role has expanded to include providing support for the Connectivity Toolkit. This team may be contacted at apisupport@towersoft.com.au, and prefers questions to include sample code where possible.

The List Serve

TOWER Software also hosts an email list for developers using the TRIM Context SDK or the TRIM Connectivity Toolkit to discuss issues arising from development with TRIM. While TOWER supports this community, we attempt to let other developers answer questions on the list first, so that this community spirit developers more effectively.

Contact the API Support team for more information on how to join the mailing list.

TOWER Helpdesk

TOWER Software runs a full time helpdesk to deal with general TRIM questions and problems. Unless you have a question specifically about the TRIM Connectivity Toolkit or the SDK, the helpdesk should probably be your first point of contact. They may be contacted via your local TOWER office.