http://iringtools.org

SDK Guide

Version 2.00.02

Notice
This is a controlled document. Unauthorized access, copying, replication or usage for a purpose other than for which it is intended, are prohibited. All trademarks that appear in the document have been used for identification purposes only and belong to their respective companies.

Document Release Note

Document Details

Name	Version no.	Description
iRINGTools SDK Guide	2.00.02	***iRINGTools*** SDK Guide

Reviewers

Name	Role	Organization Unit
Lee Colson	Documentation	***iRINGTools***
Rob DeCarlo	Software Engineer	***iRINGTools***

Terms of License

Contents

The total number of pages in this document, including the cover page, is 21.

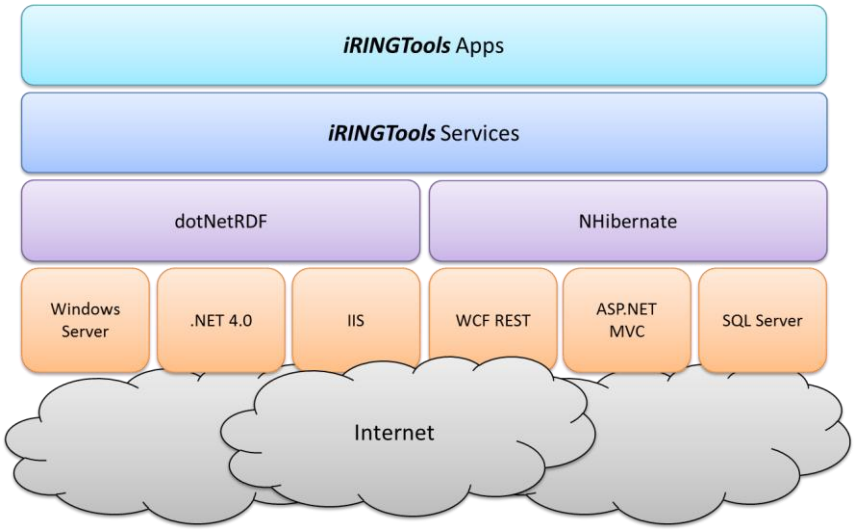
List of Abbreviations

Acronym	Description
iRING	ISO 15926 Realtime Interoperability Network Grid
ISO	International Organization for Standardization
RDSWIP	Reference Data Service Work in Progress
SP	Service Pack
GUI	Graphical User Interface
IIS	Internet Information Services
MIME	Multipurpose Internet Mail Extensions
OLTP	Online Transaction Processing
API	Application Programming Interface
CRUD	Create, Read, Update and Delete
LAN	Local Area Network
FIPS	Federal Information Processing Standard

1 Overview

iRING is a set of information interoperability and integration protocols and reference data that are compliant with the ISO 15926, Parts 7, 8, and 9 standards, which builds and depends on ISO 15926 Parts 1 through 6.

iRINGTools is a set of free, public domain, open source (BSD 3 license) software applications, services and utilities that implement ***iRING*** protocols. ***iRINGTools*** provide users with production ready deployable solutions. ***iRINGTools*** also provides technology solution providers with usage patterns for the implementation of ***iRING*** protocols in their respective solutions.

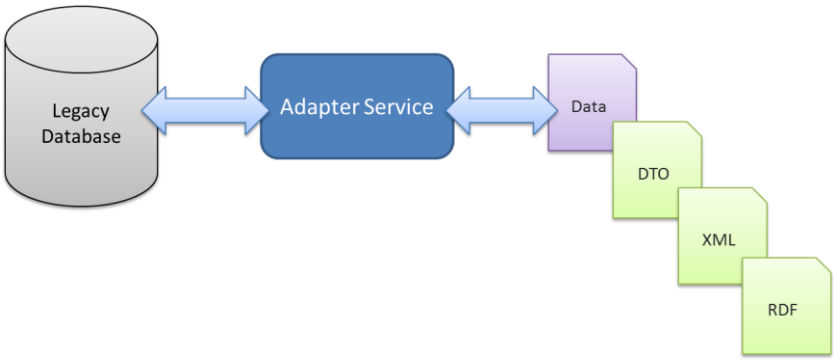


iRINGTools is deployed in two packages, ***iRINGTools*** Services and ***iRINGTools*** Apps. The ***iRINGTools*** Services are the core of the system. The ***iRINGTools*** Apps are used for mapping, configuration, and reference data management. The ***iRINGTools*** Services also support the apps, providing all information they need. The two packages are intended to be deployed into a Microsoft Windows Server 2003 running .NET 4.0 and IIS 6.0, although later versions of each are supported. The ***iRINGTools*** Apps use ASP.NET MVC, and the ***iRINGTools*** Services use WCF REST. The Services use dotNetRDF and NHibernate to handle semantic repository and relational database functionality, respectively. NHibernate and dotNetRDF both support many databases, but our sample configuration uses Microsoft SQL Server 2005 Express or later.

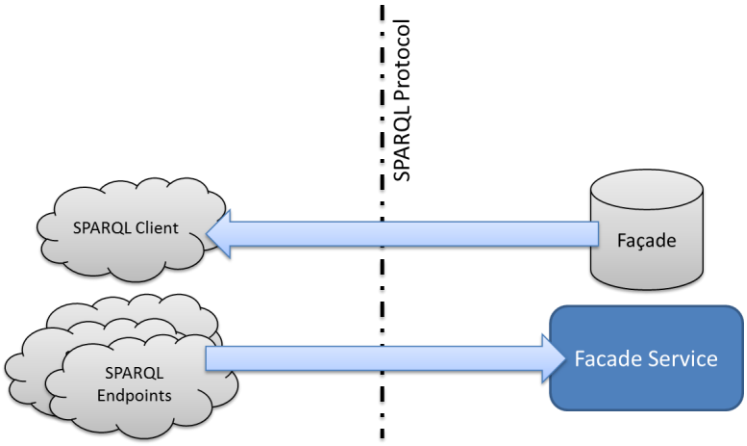
This guide focuses on the ***iRINGTools*** Adapter. The adapter is an extenisble generic web service adapter. It uses NHibernate to connect to databases for the purpose of exposing their data as a web service. Besides the default NHibernate DataLayer, the ***iRINGTools*** Adapter can be extended with a custom DataLayer. Custom DataLayers can handle other sources of data, such as an API, Excel, CSV file, etc. This document provides step by step detailed instructions for how to create and deploy a custom DataLayer with ***iRINGTools*** Adapter.

1.1 Data Exchange and Data Services

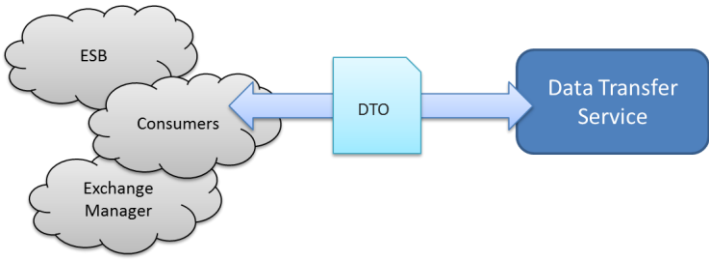
The ***iRINGTools*** Adapter can produce data projections at different levels of the ISO standard. It can produce Part 7 XML, Part 7 Data Transfer Object (DTO), Part 8 RDF/OWL, or a Part 9 Facade. The raw DataObjects are also exposed for very simple consumption needs. The purpose of exposing data in these various formats is to accommodate consumers with different needs. For the purposes of data exchange the adapter uses DTO or RDF/OWL. For general data consumption, any of the projections can be used, but the Part 7 XML is specifically optimized for this purpose. Overall, metadata is available for all levels of projection, and each format provides a uniform way of dealing with the data from various data providers.



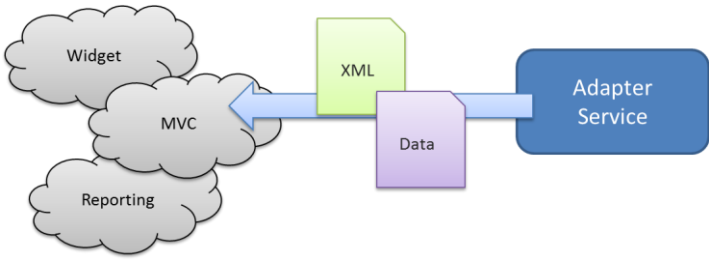
For full ISO 15926 compliance, the adapter uses RDF/OWL and SPARQL as the protocol for data exchange. In this scenario, the adapter uses pull based data exchange. This means that when the adapter receives a pull request, it can pull data from other endpoints. As shown below, the Façade Service is the SPARQL endpoint the adapter publishes to, and the SPARQL client to pull data from other SPARQL endpoints. For more information about the SPARQL protocol and SPARQL query language, please see the linked W3C documents.



The adapter can also exchange a Part 7 Data Transfer Object (DTO). The DTO is a generic object that describes the data using Part 7 Templates. The adapter uses the HTTP protocol to enable RESTful access to the DTO's. DTO's are only intended to be exchanged with other iRING endpoints that support the Data Transfer (DXFR) interface. This is not part of the ISO standard, and is not recommended for interoperability across companies. However, RESTful access to the DTO's is designed for testing and debugging purposes. It is also designed to be the easiest and fastest means of data exchange between two endpoints using Part 7 templates.



For general data consumption, the Adapter provides a data service. This service can project any of the data representations. However, for consuming Part 7 in widgets, MVC's, or reporting, a Part 7 XML is available. It is optimized for communicating relationships via simple non-generic XML. This XML has a dynamic schema, and is hierarchical. The relationships are communicated through the hierarchy.

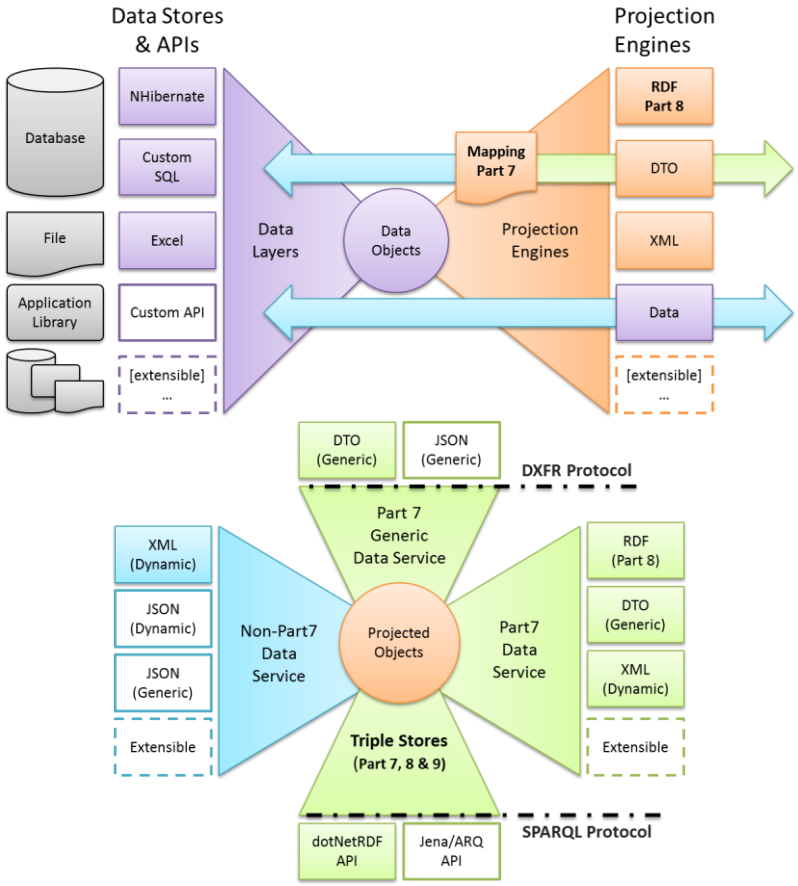


The raw DataObjects are also available for simple data structures. However, the Dictionary resource provides metadata about these objects. Also, the uniform structure allows consumers to easily deal with data from various data providers.

In future releases, the data service will provide JSON as well, and will specifically target the Sencha UI framework.

1.2 Adapter Framework

The ***iRINGTools*** Adapter is a generic and extensible service. The collection of interfaces and classes used by developers to extend the service is referred to as the Adapter Framework. The framework is centered around the data object, which is provided by the DataLayer. All of the functionality of the adapter deals with facilitating the persistence, production, consumption and projection of DataObjects. Projected objects are then exposed over multiple data APIs. Over time, new APIs can be exposed, and existing DataLayers will not need to change.



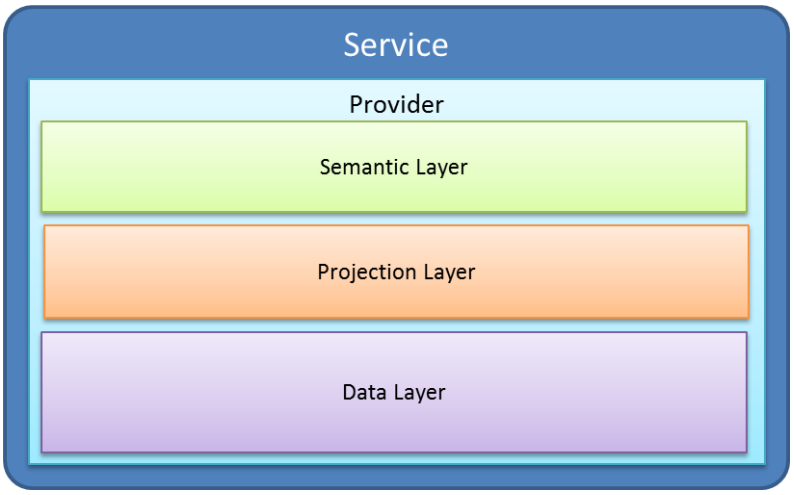
The Adapter Framework provides three primary services to a DataLayer:

- Scope and Identity
- Multiple Data APIs
- Querying and writing to a semantic repository.

1.3 Adapter Layers

The Adapter Framework is broken down into different layers which are separated by interfaces. Each layer has a defined set of functionality that it provides to the adapter.

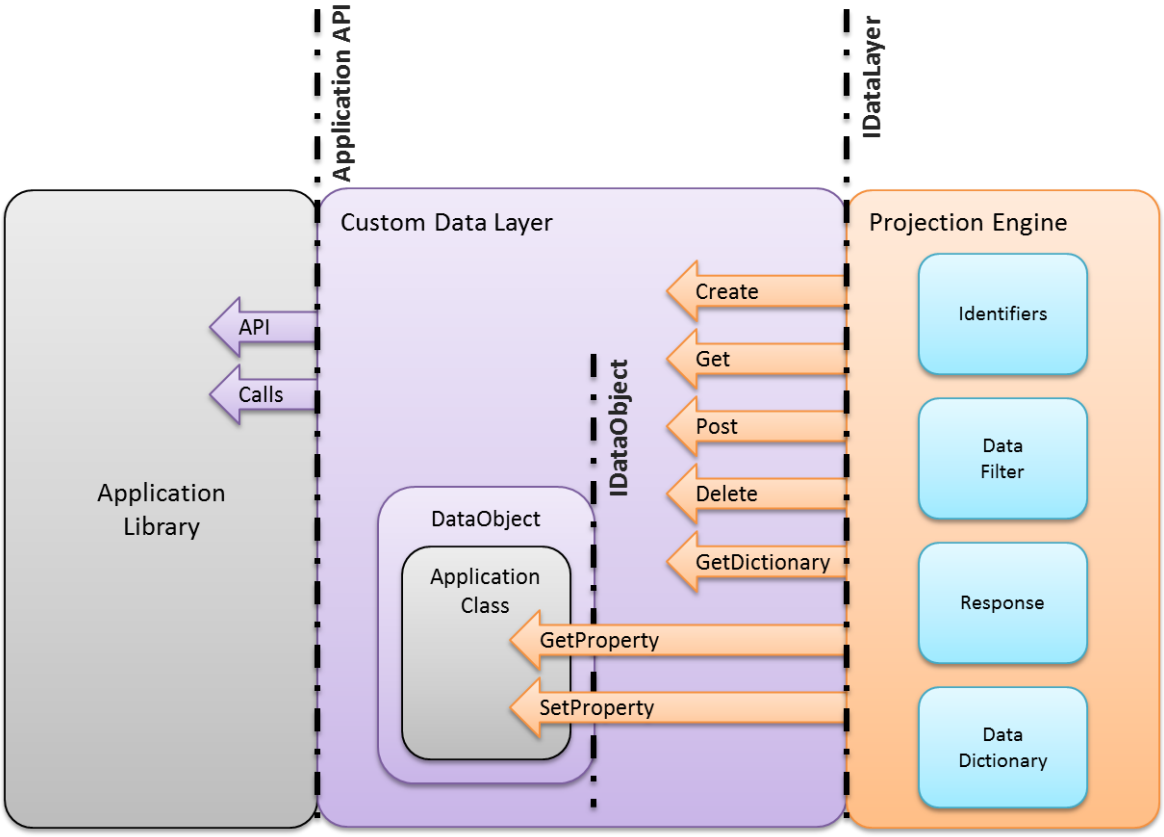
This version of the guide will focus on the IDataLayer interface. The other layers are extensible as well, and they will be documented in later versions of this guide. The IDataLayer interface consists of methods that must be implemented by each DataLayer. Through this interface, the Projection Layer will project your DataObjects based on your DataDictionary and the ***iRING*** mapping. The Projection Layer also uses this interface to create, post, and delete your DataObjects through your business logic.



This requires you to wrap your custom types in new classes that implement the IDataObject interface. In this way, the Projection Layer is abstracted from knowing anything about your custom types. The Projection Layer will simply use the DataDictionary you provide to interact with your objects through the interfaces.

2 Custom DataLayer

This section will outline the steps required to create a Custom DataLayer. Creating a Custom DataLayer involves C# development and XML configuration. The process will require the installation of the ***iRINGTools*** Services and Apps. A Custom DataLayer consists of a .NET Assembly which contains a class that implements IDataLayer.



This interface has four primary methods for Create, Get, Post and Delete. These methods work with DataObjects that implement IDataObject. The interface also has a GetDictionary method that enables the Projection Layer to get metadata about your entities. Included in the installation of the ***iRINGTools*** Adapter, are libraries that should be reference by your Custom DataLayer. These libraries provide the interfaces, and all of the classes used in them. The following sections will provide all of the details.

The steps for creating an implementation of IDataLayer are as follows:

1. Create a C# ClassLibrary Project.
2. Wrap your Entities with DataObjects that implement IDataObject.
3. Create a class that implements IDataLayer.
4. Deploy your Custom DataLayer into the ***iRINGTools*** Adapter.

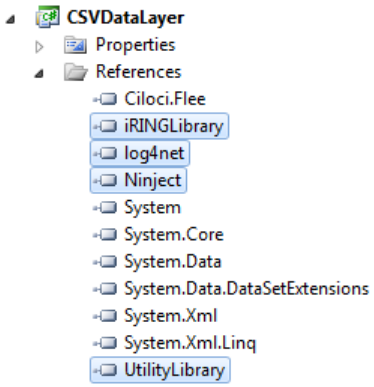
This section will guide you through each of these steps, using the sample included in the SDK.

2.1 Creating project

Create a new project of type C# ClassLibrary. Then, add a reference to the iRINGLibrary.dll and UtilityLibrary.dll from the lib/iring-tools folder in the iRINGTools-SDK-2.0.2.zip.

The Utility Library is only required if any of the utility methods are to be used. It is probably best to add it now, and decide later whether or not to use the library.

Ninject and log4net are third-party libraries that are used with the Adapter Framework. These will be required to add hooks into your code later.



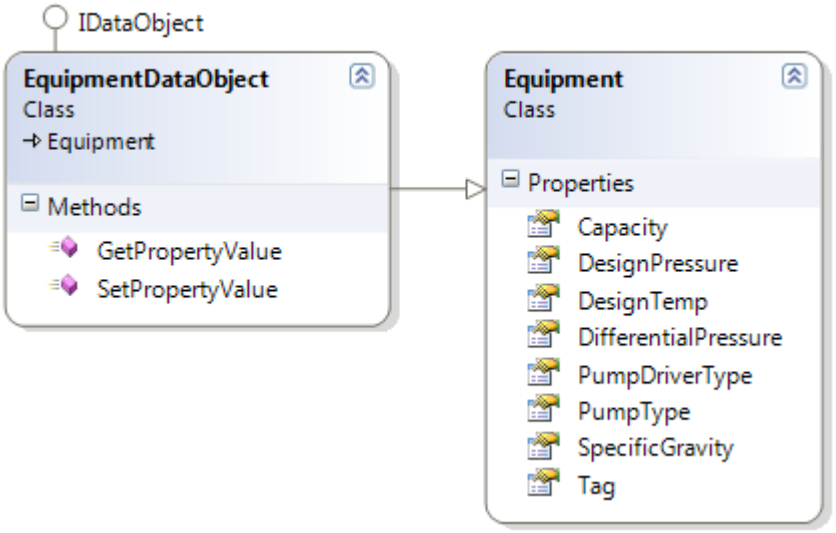
The Ciloci.Flee library is used by this specific example to enable the evaluation of LinqExpressions. This does not need to be added unless you would like this funcitonality. See the sample code for it use.

2.2 IDataObject

The IDataObject interface is how the adapter's Projection Layer interacts with your entities. Developers do not need to modify their application entities in order to implement the interface. It is recommended to wrap application entities with another class that implements the interface. Below is a listing of how this can be done.

```
public class Equipment
{
  public string Tag { get; set; }
  public string PumpType { get; set; }
  public string PumpDriverType { get; set; }
  public Double DesignTemp { get; set; }
  public Double DesignPressure { get; set; }
  public Double Capacity { get; set; }
  public Double SpecificGravity { get; set; }
  public Double DifferentialPressure { get; set; }
}

public class EquipmentDataObject : Equipment, IDataObject
{
  public object GetPropertyValue(string propertyName){...}

  public void SetPropertyValue(string propertyName, object value){...}
}
```

Below is a class diagram that shows the listing above.



The interface allows the Projection Layer to delegate the interaction with your entities to your DataLayer. Specifically, the Projection Layer uses GetPropertyValue and SetPropertyValue to access the entity properties. In this way, your implementations of IDataObject do not necessarily need to physically wrap your Entities, but just logically wrap them. This enables the flexibility to build a DataLayer around all sorts of data structures (e.g. concrete, generic, dynamic, persisted, file-based, etc).

2.3 IDataLayer

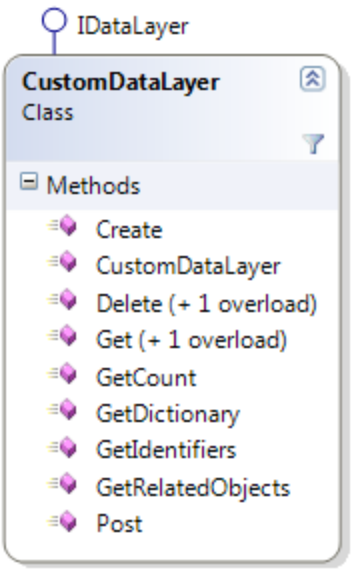
A Custom DataLayer implements the IDataLayer interface. This section will describe the steps involved in implementing this interface. Each method, and how it is used by the DTO Layer to interact with your Entities, will be explained.

To create a class that properly implements IDataLayer, the following overall steps are recommended.

1. Create Framework Hooks.
2. Implement GetDataDictionary.
3. Implement Create.
4. Implement Get, Post and Delete Operations.

The listing below shows the definition of the IDataLayer interface.

```
public class CustomDataLayer : IDataLayer
{
  public IList<IDataObject> Create(
    string objectType,
    IList<string> identifiers) {...}

  [Inject]
  public CustomDataLayer(
    AdapterSettings settings) {...}

  public Response Delete(
    string objectType,
    IList<string> identifiers) {...}

  public Response Delete(
    string objectType,
    DataFilter filter) {...}

  public IList<IDataObject> Get(
    string objectType,
    DataFilter filter,
    int pageSize,
    int pageNumber) {...}

  public IList<IDataObject> Get(
    string objectType,
    IList<string> identifiers) {...}

  public DataDictionary GetDictionary(){...}

  public IList<string> GetCount(
    string objectType,
    DataFilter filter) {...}

  public IList<string> GetIdentifiers(
    string objectType,
    DataFilter filter) {...}

  public IList<IDataObject> GetRelatedObjects(
    IDataObject dataObject,
    string relatedObjectType) {...}

  public Response Post(
    IList<IDataObject> dataObjects) {...}
}
```



For the Custom DataLayer to be used by the Adapter Framework, certain hooks need to be added. In the following sections we will cover those hooks:

1. Inject attribute
2. AdapterSettings
3. Logger
4. Response

2.3.1.1. Inject Attribute

Ninject (http://www.ninject.org/) is used to inject dependencies into the constructor of the Custom DataLayer. This is called constructor injection. It is done, by placing the **Inject** attribute on the constructor. In this way, Ninject will ensure that all of the arguments on the constructor are created before calling the constructor. There are several types of objects that may be available from the dependency injection container.

2.3.1.2. AdapterSettings

The AdapterSettings is the most useful object available from Ninject. The AdapterSettings are used in the provided sample. This parameter enables the Custom DataLayer to get the XML path and scope information that is specified by the adapter requests. It also provides identity information for authorization, which should be done by the DataLayer. Below is a table of some of the settings that may be useful.

BaseDirectoryPath	The path of the root of the running IIS Application. If you add a folder, you could use this to get your bearing.
XmlPath	The path where the ***iRINGTools*** Services store all of their XML files.
UserName	The identity of the calling user, from Windows Authentication.
KeyRing	The resulting collection of assertions from a Shared Token Service.
ProjectName	The part of the URI that specifies which project is requested.
ApplicationName	The part of the URI that specifies which application is requested.
Scope	The concatenation (with a ".") of the ProjectName and ApplicationName.

In addition to these settings, anything placed in the {XmlPath}\{Scope}.config file will be loaded and made available. Keep in mind that your settings will be ignored if they conflict with the adapter's settings.

2.3.1.3. Logger

The below code snippet shows how to create the logging hook, which will allow the DataLayer to write to the Adapter's log file.

```
private static readonly ILog _logger = LogManager.GetLogger(typeof(CustomDataLayer));
```

Once this is done, the DataLayer can use the log as in the examples below. Note the optional use of the format version of the warn method.

```
_logger.Debug("This is a debug message...");

_logger.Error("Error in Post method: " + ex);

_logger.WarnFormat("Invalid value for {0}: {1}", propertyName, value);

_logger.Info("Calling the CSVDataLayer...");
```

2.3.1.4. Response

The response object is expected to be returned by some of the methods on the IDataLayer interface. Each response contains the status for each identifier processed. Each status can have multiple messages. Additionally, the response has a messages collection for summary information.

The below code snippet shows the basic use of the response object.

```
Response response = new Response();

foreach (string identifier in identifiers)
{
	Status status = new Status();
	status.Identifier = identifier;

	//Do Work

	string message = String.Format(
		"DataObject [{0}] processed successfully.",
		identifier
	);

	status.Messages.Add(message);

	status.Level = StatusLevel.Success;

	response.Append(status);
}

response.Messages.Add("DataObjects processed successfully.");
```

The setting the status level consistently with others is very important. Errors should only be unexpected errors that occur in your code. Warnings should include any restrictions due to business rules, or other warnings. The level on each status is set independent of the other status entries. When it is appended, the Response object will manage the overall Level to ensure that the most critical level is always shown to the user.

2.3.2 GetDataDictionary Method

The DataDictionary provides metadata which is used by the Projection Layer, which does all of the interaction with the DataLayer. It is a class that defines DataObjects to be exposed to the adapter and ultimately exchanged with other applications.

It is recommended that it be created programmatically. To create a DataDictionary programmatically, you need to create an instance of DataDictionary with DataObjects and optionally data relationships. The tests will write your DataDictionary to XML, so that you can review it. The listing below shows a GetDictionary implementation that does this.

```
// Create a DataDictionary instance
DataDictionary dataDictionary = new DataDictionary()
{
  dataObjects = new List<DataObject>()
  {
    new DataObject()
    {
      keyDelimeter = "",
      keyProperties = new List<KeyProperty>()
      {
        new KeyProperty()
        {
          dataLength = 50,
          dataType = DataType.String,
          propertyName = "Tag",
          isNullable = false,
          keyType = KeyType.assigned,
        },
      },
      dataProperties = new List<DataProperty>()
      {
        new DataProperty()
        {
          dataLength = 255,
          dataType = DataType.String,
          propertyName = "PumpType",
          isNullable = true,
        },
        ...
        new DataProperty()
        {
          dataLength = 16,
          dataType = DataType.Double,
          propertyName = "DifferentialPressure",
          isNullable = true,
        },
      },
      objectName = "Equipment",
    }
  }
};

return dataDictionary;
```

The identifiers argument used in the interface is a list of identifiers. Each identifier is a concatenation of the key properties specified in the DataDictionary. The keyDelimiter is used to perform this concatenation. It is important to note that each data object must have at least one KeyProperty for the ***iRINGTools*** Adapter to identify each individual data object. However, it is best to use a single identifier whenever possible, like a number. This identifier will only be used on the direct DataObject projection.

2.3.3 Create Method

The create method is used by the DTO Layer to get empty instances of DataObjects to fill with data and post back to the custom DataLayer. The create method should create empty (except for the identifiers) instances of DataObjects, and return an IList<IDataObject>, which is a list of instances with the specified identifiers. Changes to your internal system, the application of business rules and authorization should not be done in this method. The DataLayer is simply creating an instance of the Data Object on behalf of the Projection Layer.

2.3.4 Get, Post, and Delete Methods

The Get, Post and Delete methods are the core of the interface. They should apply business logic, perform authorization, and modify the internal system, if appropriate. The details of what should be done in each method is described below.

2.3.4.1. Get Methods

The **Get** method should provide data to the Projection Layer for projecting into the various formats. It returns a list of IDataObjects.

There are two overloads of this method. One that takes the objectType and filter (a pagesize and a startIndex can also be specified here). Another that takes the objectType and list of identifiers. The Filter is a structured filter definition that can be used to create whatever filtering logic or syntax is required by your application. The iRINGTools Utility Library provides a method for converting this into a SQL WHERE clause if that is desired. The sample shows how to use a similar utility to form LinqExpressions.

To return a list of DataObjects, data needs to be fetched from the application and marshalling can be done to marshal application's specific datatypes into IDataObjects.

The GetCount and **GetIdentifiers** are similar, but should provide either a simple count or a list of identifiers (strings), respectively. These are used by the adapter to facilitate different web APIs.

The **GetRelatedObjects** method should provide data to the Projection Layer for projecting into the various formats. It returns a list of IDataObjects of the specifies relatedObjectType that are related to a provided dataObject .

This puts the DataLayer in control of how these two DataObjects are related and how to join them. The Projections Layer uses the relationships specified in the DataLayer to know when to call this method.

2.3.4.2. Post Method

Post requests should receive data from the Projection Layer, and a detailed response should be returned for each IDataObject. This detailed response should include success, warnings, errors, or a detailed rejection explanation.

The data received needs to be inspected to determine if it is an add or change operation and the marshalling may be required to marshal DataObjects into application's specific datatypes.

2.3.4.3. Delete Method

Delete method should delete DataObjects after verifying that it exists based on objectType and filter or based on the objectType and list of identifiers. A detailed response should be returned for each IDataObject. This detailed response should include success, warnings, errors, or a detailed rejection explanation.

2.4 Deploying the Custom DataLayer

Once you build the Custom dataLayer, the next step is the scope configuration. Please see the User Guide for instructions on how to add a scope. Once this configuration is done, the next step is to hook its DLL into the ***iRINGTools*** Adapter. In order for ***iRINGTools*** Adapter to use a custom DataLayer, the custom DataLayer must be bound to the IDataLayer interface. The details of how to do this will be described in detail below.

Binding configuration XML is located at {XmlPath}\BindingConfiguration.{Scope}.xml. It contains one binding for DataLayer. This specifies which implementation of the DataLayer should be used for an application. It is loaded by the Framework once per WCF service request and stored in Ninject session so that any reference to the DataLayer will get resolved dynamically at runtime.

In CSVDataLayer, the binding configuration XML is as follows. Note that the second part after the comma in the Interface and Implementation definition are assembly names.

```
<?xml version="1.0" encoding="utf-8"?>
<module name="12345_000.CSV">
  <bind name="DataLayer"
        service="org.iringtools.library.IDataLayer, iRINGLibrary"
        to="iRINGTools.SDK.CSVDataLayer.CustomDataLayer, CSVDataLayer" />
</module>
```

The BindingConfiguration is the last step in setting up your DataLayer. The AdapterService should now respond to requests for your data. The data service can be used to directly test your data layer without any mapping. However, it is read only, you will need to perform some mapping to test Post and Delete functionality.

3 Example Implementation

This document provides an example of a custom DataLayer used in iRINGTools demonstrations, called CSVDataLayer, which reads comma-separated values (CSV) data from disk, turns it into DataObjects, and writes it back to disk as its original CSV format. The complete CSVDataLayer source code and binary files mentioned in the document are available in the SDK package as a zip file.

Unzip the **iRINGTools-SDK-2.0.1.zip**.
Open **iring-tools-sdk.sln** in VisualStudio 2008 or later.

The following components will be present in the project for your reference:

- CSVDataLayer.cs that implements IDataLayer interface.
- DataObjects.cs that has the Equipment entity and EquipmentDataObject that wraps the Equipment entity and implements IDataObject interface.
- Equipment.12345_000.API.xml that contains the order of attributes of the csv file.
- 12345_000.API\Equipment folder that contains the sample csv files.