Summer Cart
Synchronization Guide for .NET

Introduction

This guide explains how you can synchronize the data from your data management software with your Summer Cart-based web store. In the ZIP package you will find the source code of an example application written in .NET that demonstrates a synchronization of an SQL Server 2005 Express database with a Summer Cart store. If you read the guide and study the example application, you will be able to customize it in a way so you can synchronize the data from your own software, no matter what type of database you use and how your data is organized.

How It Works

Summer Cart allows for data synchronization with other systems through web services. Any system that is given a Summer Cart Sync API account can manage the content of the store through a convenient platform- and technology-independent API. Mirchev Ideas has developed a .NET library (SummerCart.SynchronizationService.dll) that further streamlines and simplifies the synchronization process, and a sample application is provided that demonstrates the usage of the library in a real world scenario.

Prerequisites

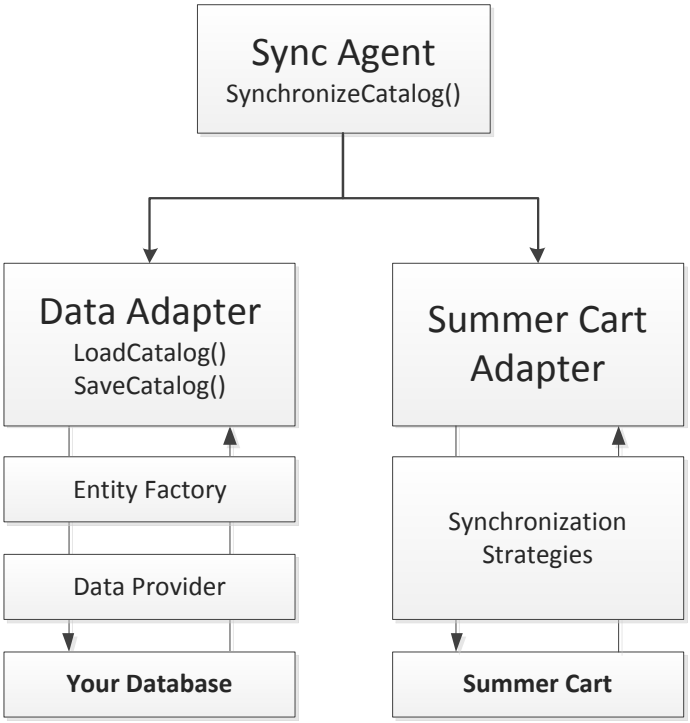
Before you begin, make sure you have the following prerequisites installed:

- .NET Framework 3.5 or later
- SQL Server 2005 Express Edition (http://go.microsoft.com/fwlink/?linkid=65212)
- A Summer Cart installation with a Sync API account – see the “How Do I?” section for more information on how to create an API account.
- Visual Studio 2008+ or compatible development environment

Contents

Getting Started

The demo application uses Sync Agents for synchronization. A **Sync Agent** we call any class that implements ISyncAgent, an interface with a single method called SynchronizeCatalog(). This method is supposed to synchronize your catalog[1] and your Summer Cart web store. The implementation of ISyncAgent that ships with the demo application is capable to synchronize data that comes from any type of database supported by .NET and a Summer Cart store. For this to work, the sync agent uses two adapters: a Data Adapter and a Summer Cart Adapter. The **Data Adapter** works with your database. It has only two methods, LoadCatalog() and SaveCatalog(). It does not directly work with raw database queries, but instead uses two other classes. A **Data Provider** contains the SQL queries necessary to work with your database and returns DbReader instances. An **Entity Factory** is able to convert the raw data coming from a DbReader to business entity. Finally, a **Summer Cart Adapter** is what synchronizes the catalog loaded by the Data Adapter with your Summer Cart web store. It uses a number of **Synchronization Strategies**, each one knowing how to synchronize a specific business entity.



[1] In Summer Cart, "catalog" is a collective name for all the business data that is available in a store, such as categories, manufacturers, products, customers, etc.

You may be wondering why we need all those classes. In fact, these are all interfaces: we have ISyncAgent, IDataAdapter, IEntityFactory, IDataProvider and so on. All of them contain only the basic set of properties and methods needed, and you can provide different implementation for any of the interfaces. For example, if you don't use SQL Server, but e.g. Oracle or MySQL, or if your table structure is different than the one in the demo application, you would only need to provide an alternative implementation of your Data Provider and maybe for your Entity Factory. Everything else will stay the same.

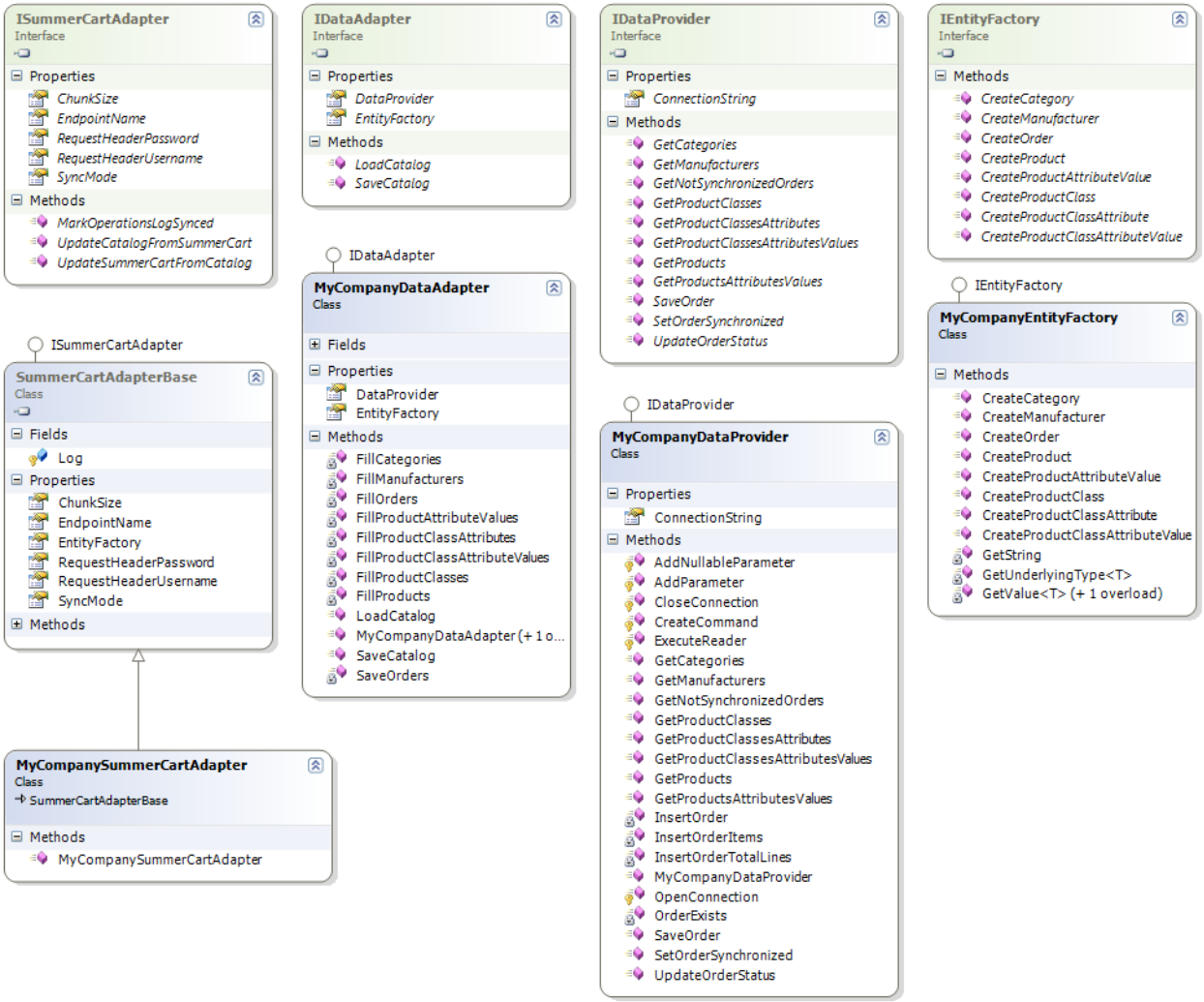
A chapter later in this guide, called "Modifying the Sample Application," explains exactly what you have to do to modify the demo application so it works with your data source.

If you open your App.config file, you will see something similar to the following:

```
<syncAgents>
    <syncAgent name="MyCompany">
      <dataAdapter
        type="MyCompany.SummerCartSynchronization.Adapters.MyCompanyDataAdapter, …"
        connectionString="<Your Connection String>"/>
      <summerCartAdapter
        type="MyCompany.SummerCartSynchronization.Adapters.MyCompanySummerCartAdapte"
        endpointName="MyCompanySummerCartServiceEndpoint"
        syncMode="Strict"
        username="<Your Sync API Username>"
        password="<Your Sync API Password>"
        getSyncChunkSize="5000"
        modifyChunkSize="1000"/>
    </syncAgent>
</syncAgents>
```

Here, we define a single Sync Agent called "MyCompany". We specify that the type of its data adapter is MyCompany.SummerCartSynchronization.Adapters.MyCompanyDataAdapter. If we needed another implementation of the data adapter, we could have specified it directly from the App.config file. We also set the connection string to the database. Similarly, we specify the type of the Summer Cart adapter, which endpoint to use (this should be changed to match the address of your store), Sync API username and password, and chunk sizes. Later in this guide these properties will be explained in greater detail.

The following diagram illustrates the 4 basic interfaces and their default implementations:



Running the sample application

To run the sample application, follow these steps:

1. Open “MyCompany.SummerCartSynchronization.sln” in your Visual Studio;
2. In your App.config, replace “www.mycompany.com” with the actual address of your web store;
3. In your App.config, modify the <summerCartAdapter> element so it uses the username and password of your API account;
4. Attach Data\MyCompany.mdf to an SQL Server of your choice. By default it is set to use .\SQLEXPRESS. If you attach the database to another server, modify the <dataAdapter> element set a correct connection string.
5. Run the application.

Command line options

The sample application supports these command line arguments:

- **-sync** Tells the program to perform synchronization;
- **-agent:*AgentName*** Tells the program to use an agent specified by *AgentName* for synchronization. An agent with such name must be defined in the configuration file;
- **-contents:*Contents*** Tells the program to synchronize only the specified catalog contents. You must provide an integer number that can be converted to the enum CatalogContents. For example, if you want to synchronize only Categories and Manufacturers, specify 1 + 2 = 3. Default is CatalogContents.All.

These arguments are available to the Main method in an object-oriented way through the StartupParameters class. Note that if you run the program with no arguments, nothing will happen. If this is not the desired behavior, you can always modify your Main method to ignore parameters.Synchronize or the StartupParameters class to set Synchronize to true by default.

Understanding how the sample application works

This is the sequence of events that takes place in the application when you run it:

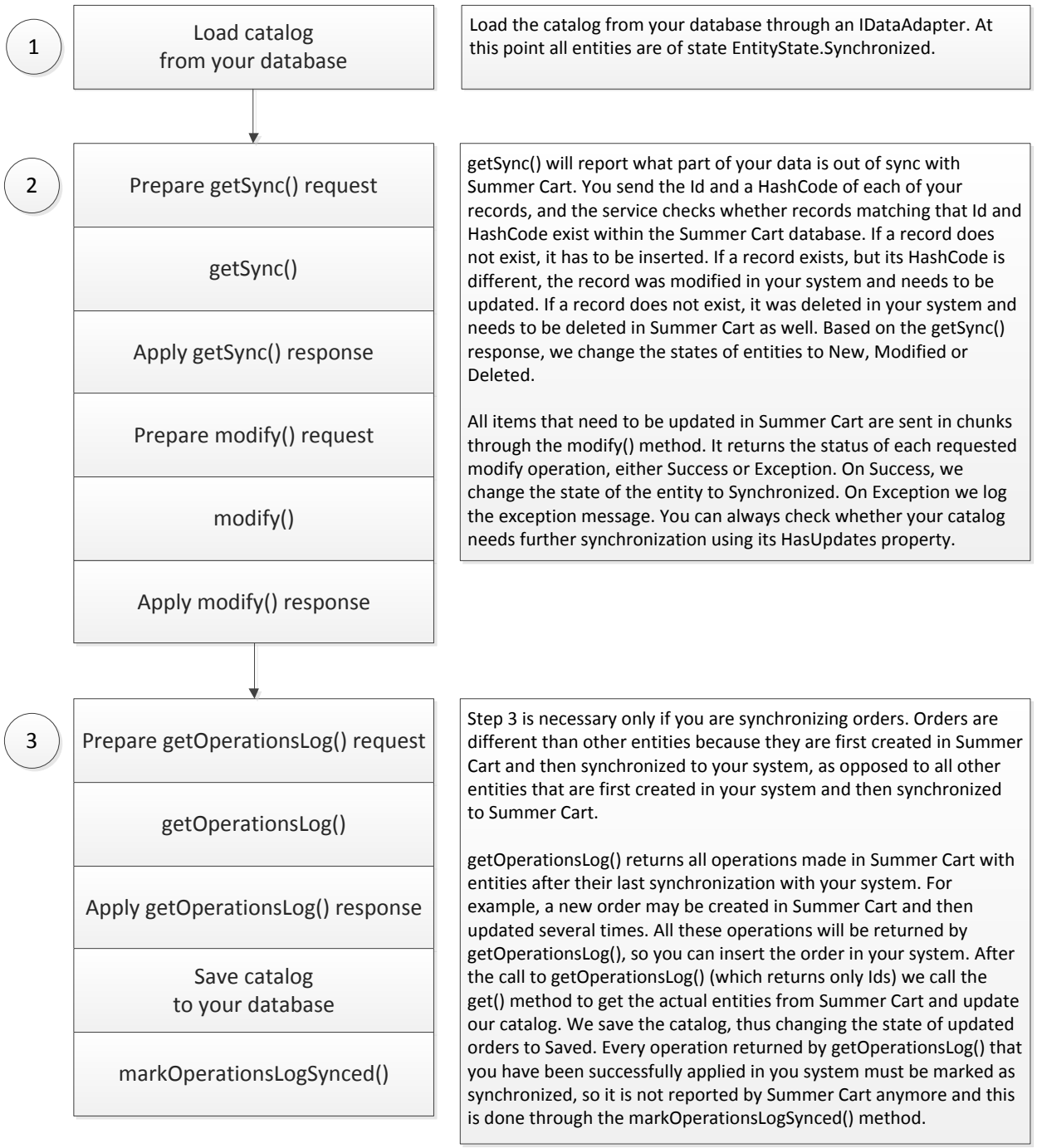
1. The Main method processes your command line arguments. By default in Debug mode these arguments are "–sync –agent:MyCompany". Then an ISyncAgentFactory instance of type SyncAgentFactory is created. This factory is responsible for creating sync agents. It would create sync agents of type SyncAgent, which is the default implementation. A sync agent will be created based on the information specified in your App.config file. Here, if you need a custom ISyncAgentFactory, you can implement one and replace this line with one that uses your custom factory. A custom factory may create custom sync agents that, for example, are not configured through App.config, or do not work with a database, or work with several Summer Cart shops, or any other advanced scenario that you can think of.

2. The Main method asks the sync agent to synchronize the catalog. You may want to synchronize the whole catalog at once using CatalogContent.All, or synchronize only part of the catalog. The demo application shows how to synchronize the whole catalog, but in 7 steps – manufacturers first, then product classes, then attributes and so on. This would be a better approach if you have a lot of data. See the Best Practices section for further details on how to best organize the synchronization.

3. The sync agent will load your catalog, then first update Summer Cart with your latest changes, then update your catalog with Summer Cart's changes (currently only orders) and finally save your catalog.

You can find all the diagrams from this guide, along with others, in the Diagrams folder of the demo application.

This is the sequence of actions performed by the sync agent to synchronize your catalog:



Summer Cart 4 added support for sync transactions. Until this version the Summer Cart adapter had to send the Id's and hash codes of all your business entities of a particular type (e.g. products) in one call to the service, using the getSync() method. This required you to increase script and message sizes if you have a large number of entities, e.g. over 50,000. The latest version of the adapter internally uses the new transaction mechanism. First it opens a sync transaction (the startSyncTransaction() method), then it adds sync info in chunks whose size is specified by the getSyncChunkSize attribute in your App.config (the addSyncInfo() method), then it commits the transaction (the performSyncTransaction() method) and finally gets and applies the results to your catalog, again in chunks (the getSyncTransactionResults() method). This change in the internal usage does not affect the way you would be running, using, or modifying the demo application.

Modifying the Sample Application

Replace "MyCompany" with your company name

Your first step should be to replace MyCompany with your company name around the project. For this you could use the Replace in Entire Solution feature of Visual Studio, but you will have to rename some files by hand. Don't forget to change the command arguments in Debug mode from "–sync –agent:MyCompany" to such that use your agent name.

Modify your App.config

In your App.config, replace "www.mycompanystore.com" with the actual URL of your web store, and modify the sync agent so it uses your API username and password, and a connection string to your system.

Modify the MyCompanyDataProvider class

MyCompanyDataProvider implements the methods defined in the IDataProvider interface, such as GetCategories(), GetManufacturers() and so on. It is supposed to return DbDataReader instances that fetch the various types of data in a catalog. This is the only class in the system that is specific to the database vendor. If you're not using SQL Server, you should change the OpenConnection method so it uses another provider (e.g. OracleConnection, OdbcConnection, etc.). Modify the various Get() methods so they use queries that return the data you want to synchronize. The sample application has a database structure that exactly matches that of Summer Cart. Granted, your structure will be different, so you can either specify your (probably complex) SQL queries directly in the code (as in the sample application), in a resource file, or use views or stored procedures in your database that adapt the data. Of course these are just some of the possible approaches.

Modify the MyCompanyEntityFactory class

MyCompanyEntityFactory implements the methods defined in the IEntityFactory interface, such as CreateCategory(), CreateManufacturer() and so on. This is the place to convert the data that comes from your database to entities. In the sample application this is a really trivial task, as the database structure matches exactly that of the entities. In your application you'd probably need to write some more code. Note how all Id properties of all entities are of type string. This is so Summer Cart supports any type of primary keys because anything (ints, longs, guids, etc)

can be converted to string. Note that in addition to the MyCompanyEntityFactory, which converts from your database model to the model of the synchronization application, there is one more entity factory called SummerCartEntityFactory, which converts from the Summer Cart model to the model of the synchronization application. This is used only for entities that are synchronized from Summer Cart to your system, e.g. orders. You would not need to understand, implement, or in fact use this factory at all from your code unless you have a custom Summer Cart store with custom orders.

Modify the MyCompanyDataAdapter class

MyCompanyDataAdapter implements the methods defined in the IDataAdapter interface: LoadCatalog() and SaveCatalog(). You are free to load and save the catalog in any way you want, but a common scenario is implemented in the sample application. You have various Fill() methods, such as FillCategories(), FIllManufacturers(), which fill the respective properties of the catalog, such as Categories, Manufacturers and so on. For entities that you are supposed to save in your database (e.g. orders) there are Save() methods such as SaveOrders(). Each of these methods uses the underlying IDataProvider to get the raw data, and then uses the underlying IEntityFactory to construct entities based on the raw data. The DataAdapter class is the place to perform any custom entity processing logic, validation and sorting. The sample application provides an example on how to sort Manufacturers alphabetically instead of using the SortIndex from the database.

Modify the MyCompanySummerCartAdapter class

MyCompanySummerCartAdapter inherits from SummerCartAdapterBase, which implements the ISummerCartAdapter interface. Unless you're dealing with a custom Summer Cart service you won't need to make any modifications to this class. For easier debugging you may want to override the public methods of the class with identical implementations from the base class. This way you can set breakpoints and go through the entire synchronization process. The source code of SummerCart.SynchronizationService.dll is available upon request.

Remove the sample database

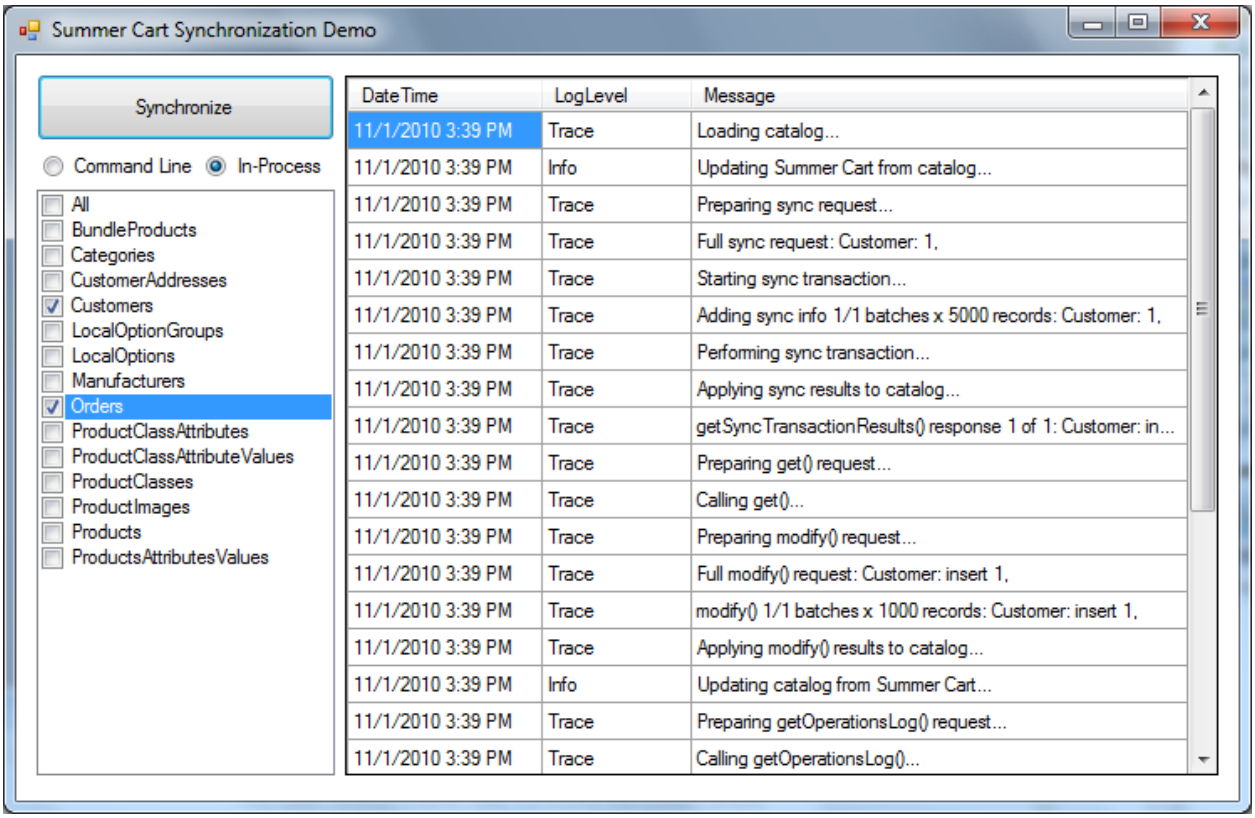
As soon as you have tested the application with the sample SQL 2005 Express database, you can delete it.

Working with custom Summer Cart services

If you are working with a custom Summer Cart service (i.e. one customized for a specific client) you will have to reference the SummerCart.SynchronizationService project directly, configure your service references and modify the Summer Cart adapter and/or entities to fit your needs. The source code of SummerCart.SynchronizationService.dll is available upon request.

Demo Application

As a demonstration of how you can call the MyCompany application from another program (be it in .NET or otherwise) and process/display synchronization messages, we have developed an additional demo application that has a simple GUI.



The demo application allows you to synchronize your full catalog or select and synchronize only the entities you need. It demonstrates two ways to communicate with the sync application:

- **Command Line** The sync application is called as a separate process, and correct command line parameters are passed. Its NLog.config file is configured with a Console target so all messages are printed in the console, and the console output itself is redirected to the demo application, which processes the messages and displays them in a grid. (No console window is visible to the user.) This method of synchronization works best if your calling application is not written in .NET, but you still want to process synchronization messages;

- **In-Process** The sync application is added as a reference to the calling application and its Main method is called directly. This way the sync application is executed in the same process as the calling application. The NLog.config file is configured with a MethodCall target so all messages are directly sent to a method of the calling application, which processes them and displays them in a grid. This is the preferrable method if your calling application is written in .NET.

Best Practices

Please consider the following best practices when your build your Summer Cart synchronization application:

- Validate your data before sending it to the service. At database level, use unique primary keys and foreign keys to assure referential integrity. Failing to provide the service with valid primary keys and foreign keys will result in sometimes obscure error messages. It will be much easier to prevent incorrect data to reach the service than figuring out what is wrong from verbose log files.

- Based on the amount of your data, you may consider calling the service in several passes. This is very straightforward to use. This is how we synchronize all items at once:

  ```
  syncAgent.SynchronizeCatalog(CatalogContents.All);
  ```

  This is how the same thing can be done in multiple steps, each time synchronizing just a part of the catalog:

  ```
  syncAgent.SynchronizeCatalog(CatalogContents.Manufacturers | CatalogContents.Categories);
  syncAgent.SynchronizeCatalog(CatalogContents.ProductClasses);
  syncAgent.SynchronizeCatalog(CatalogContents.ProductClassesAttributes);
  syncAgent.SynchronizeCatalog(CatalogContents.ProductClassesAttributesValues);
  syncAgent.SynchronizeCatalog(CatalogContents.Products);
  syncAgent.SynchronizeCatalog(CatalogContents.ProductAttributeValues);
  ```

  The reason you may want to do this is because of memory limits configured in your hosting environment. Our tests indicated that a standard Summer Cart installation on a server that grants up to 8 MB of memory to a script, no more than about 10000 records can be imported by the script on a single pass. If you have more than 10000 records in your database, you will need to either synchronize on several passes or increase the maximum amount of memory given to scripts.

Error Handling

The sample application demonstrates the usage of log files and console to output the result of the synchronization, although you can change this behavior easily. We are using the popular open-source logging framework NLog for all logging purposes. The sample application demonstrates the usage of the following logging rules:

- All messages with level Debug or above are displayed in the console window;
- All messages with level Debug or above are logged into /Logs/Session.log. The lifetime of this file is for the session only, it is overwritten if you run the application again;
- All messages with level Error or above are logged into /Logs/Errors.log. This file is NOT overridden on next startup, so you always have a full history of application errors.

These rules are configured in the file NLog.config and are not hard-coded into the application. You can change the rules and the logging behavior completely by only editing the NLog.config file. For example, you may want e-mail messages to be sent to specific users when certain event happens. You can also handle events in a completely custom way by your own code using a MethodCall target. In this target you specify method handler(s) for events and NLog will invoke them when the specified events are raised. Note that you can handle events from different sources in a different way. For example, an error event from the Program class should typically abort your synchronization process, and the user should be notified about the error, while errors from your data adapter (indicating incorrect data in your database) may be ignored and maybe only a list of warnings displayed to the user.

Consult NLog documentation for more detailed information on how to use and configure NLog:
http://nlog-project.org/wiki/Documentation

More information about the MethodCall target can be found here:
http://nlog-project.org/wiki/MethodCall_target

Log Levels

The sample application and the Summer Cart synchronization library both log messages with different log levels based on message importance and verbosity. Following is a list of used log levels and when they are used:

- **Trace** – log messages with maximum details. Full entity contents and complete requests and responses are logged. By default trace messages are not logged as they are rather verbose. You can easily enable their logging by modifying the min log level of your targets in your NLog.config file.

- **Debug** – debug messages contain summary information on what is being synchronized. For example, when you modify your store, debug messages will tell how many entities are inserted, updated or deleted, but will not tell the exact contents of the entities (you will have to enable Trace logging to find this out).

- **Info** – informational messages that are not programming specific and may be displayed to the user.

- **Warn** – warning messages that may or may not be displayed to the user, or alternatively sent via email to developers on handled in another way.

- **Error** – errors that would not stop or crash the application. For example, if a single entity could not be synchronized, an error message will be logged, but the application will continue with the next entity. Error messages may or may not be displayed to the user, or alternatively sent via email to developers on handled in another way.

- **Fatal** – errors that prevent the synchronization process from continuing and cause the application to immediately exit. Fatal error messages may or may not be displayed to the user, or alternatively sent via email to developers on handled in another way.

Troubleshooting

There are several things that may go wrong with your synchronization:

- If your SQL queries are invalid or there is a problem with your connection string, you will get error messages pointing to your DataProvider;

- If you are not constructing your entities correctly, you will get error messages pointing to your EntityFactory;

- If your data makes it to the service, you can either receive an error message from the getSync() operation, or from the modify() operation. In some scenarios the message will not be logged by NLog, and you will have to debug your WCF service logs. Look at your Logs folder for the files “WCF.Messages.svclog” and “WCF.Trace.svclog”. You can open them using the Microsoft’s Service Trace Viewer. To turn on WCF service logging, open

your App.config file, locate the <messageLogging> element within <diagnostics>, and change logEntireMessages, logMalformedMessages and logMessagesAtTransportLevel to “true”.

- You can use the built-in ObjectLogger to conveniently log whole objects, such as your catalog, requests and responses from getSync() and modify().

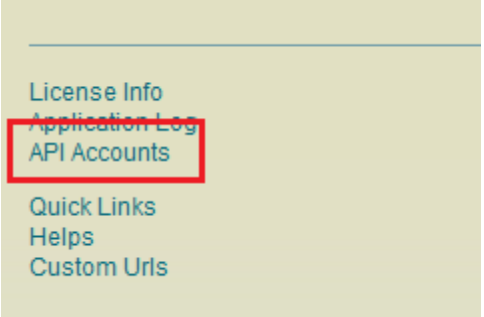
```
LogObject("SyncRequest.log", request);
service.getSync(header, request, out response);
LogObject("SyncResponse.log", response);
```

Please read the Best Practices section for practices that will greatly reduce the time you spend debugging and troubleshooting.

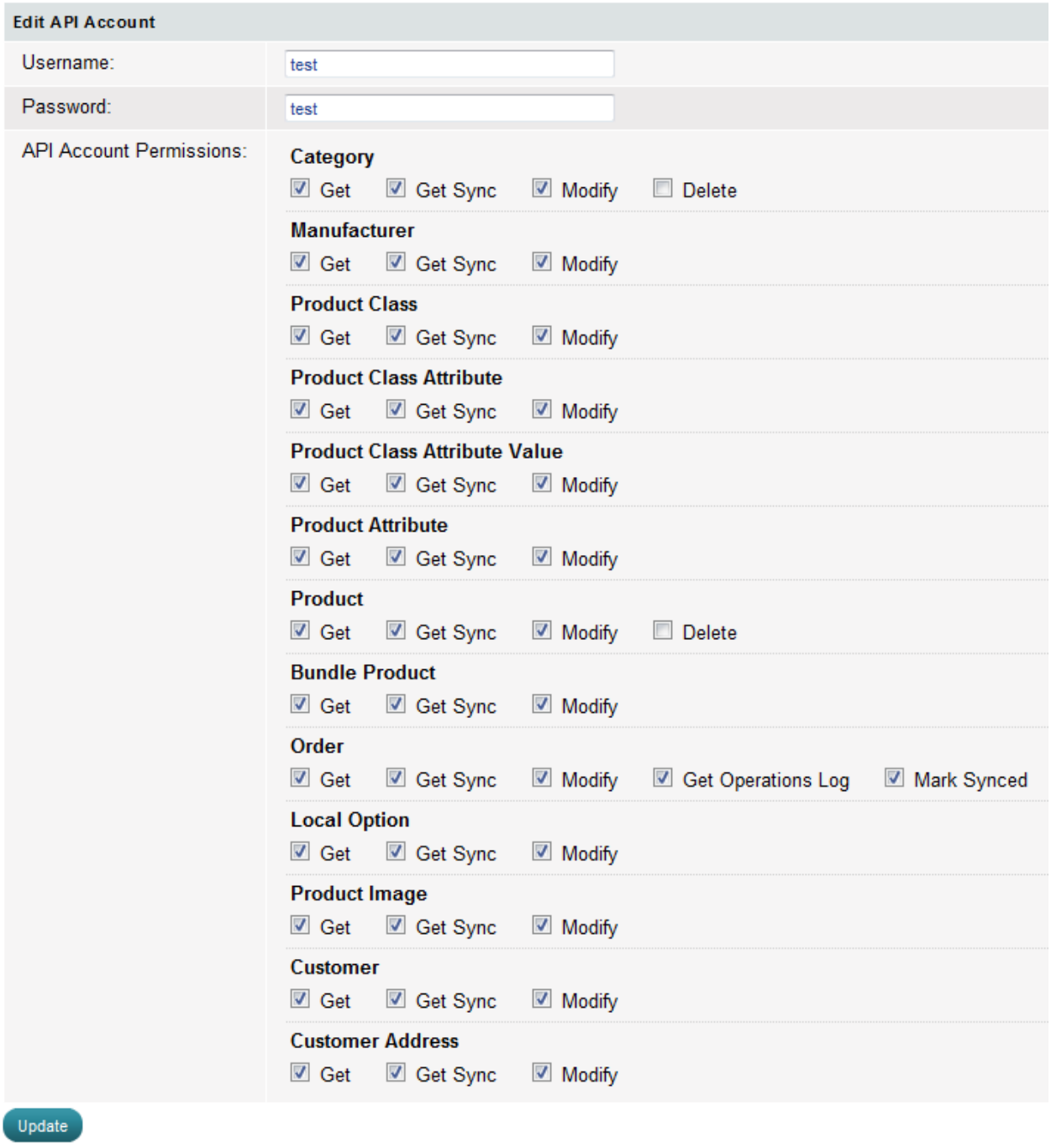
How Do I?

Create an API account in Summer Cart?

1. Open your Summer Cart Admin panel.
2. Navigate to the bottom of the page and click API Accounts



3. Add a new API account (*see next page for screenshot)*. For each data type that can be synchronized, you can specify if the API account has access to the get(), getSync() and modify() methods of the service in regards to this data type.


Edit API Account
Username:
test
Password:
test
API Account Permissions:
Category
Get
Get Sync
Modify
Delete
Manufacturer
Get
Get Sync
Modify
Product Class
Get
Get Sync
Modify
Product Class Attribute
Get
Get Sync
Modify
Product Class Attribute Value
Get
Get Sync
Modify
Product Attribute
Get
Get Sync
Modify
Product
Get
Get Sync
Modify
Delete
Bundle Product
Get
Get Sync
Modify
Order
Get
Get Sync
Modify
Get Operations Log
Mark Synced
Local Option
Get
Get Sync
Modify
Product Image
Get
Get Sync
Modify
Customer
Get
Get Sync
Modify
Customer Address
Get
Get Sync
Modify
Update