

# Dynamics AX for Retail Sdk and Extensibility Handbook

## Disclaimer

Microsoft Dynamics is a line of integrated, adaptable business management solutions that enables you and your people to make business decisions with greater confidence. Microsoft Dynamics works like and with familiar Microsoft software, automating and streamlining financial, customer relationship and supply chain processes in a way that helps you drive business success.

U.S. and Canada Toll Free 1-888-477-7989

Worldwide +1-701-281-6500

[www.microsoft.com/dynamics](http://www.microsoft.com/dynamics)

© 2016 Microsoft Corporation. All rights reserved. This document is provided "as-is." Information and views expressed in this document, including URL and other Internet Web site references, may change without notice. You bear the risk of using it. Some examples are for illustration only and are fictitious. No real association is intended or inferred.

This document does not provide you with any legal rights to any intellectual property in any Microsoft product. You may copy, modify and use this document for your internal, reference purposes.

# 1 Contents

1	Overview .....	6
2	Structure of this document .....	7
2.1	Abbreviations .....	7
3	Retail Sdk overview .....	8
3.1	Rapid development .....	8
3.2	Full MSBuild integration .....	8
3.3	Packaging tools .....	8
3.4	Facilitate better code separation .....	8
3.5	Real-world implementation samples .....	8
4	Retail Sdk deep dive .....	9
4.1	Retail Sdk checklist .....	9
4.2	Obtaining the Retail Sdk .....	9
4.3	Prerequisites .....	10
4.4	Retail Sdk contents .....	10
4.5	Retail Sdk Extensibility samples and tools .....	12
4.6	Dependencies, build order and full build .....	14
4.7	Minimal configuration .....	17
4.8	Normal configuration/code signing .....	17
4.9	Adding a payment connector .....	19
4.9.1	Adding the files .....	19
4.9.2	Register payment accepting page URL .....	19
4.9.3	Register payment device assembly .....	20
4.10	Customizing the build .....	20
4.10.1	Adding new projects .....	20
4.10.2	Changing build order or adding to the build .....	22
4.10.3	Build script customization .....	22
4.11	Developer activities (update, run, debug, etc.) .....	23
4.11.1	One-time RetailServer setup for rapid development .....	23
4.11.2	Debugging RetailServer and CRT code .....	24
4.11.3	Debugging ModernPOS with F5 from Retail Sdk .....	24
4.11.4	Debugging CloudPos with F5 from Retail Sdk .....	25
4.11.5	Verifying deployment and customizations of ModernPOS in offline mode .....	26

4.11.6	Using the RetailServer Test Client.....	27
5	Application lifecycle management (ALM) .....	29
5.1	Branching and versioning.....	29
5.2	Retail Sdk mirror branch .....	29
5.3	Customization branch .....	30
5.4	Branching walkthrough for Visual Studio Online.....	30
5.4.1	Sign into Visual Studio Online and verify server and project connectivity .....	30
5.4.2	Create the Retail Sdk mirror branch and add the unchanged files.....	30
5.4.3	Create the customization branch.....	33
5.4.4	Updating the Retail Sdk mirror branch .....	34
5.4.5	Updating/merging the customization branch with an updated Retail Sdk mirror .....	35
6	Getting started writing your customizations .....	36
6.1	Best practices .....	36
6.1.1	Use of source control .....	36
6.1.2	Location of code changes.....	36
6.1.3	Use Customization.settings.....	36
6.2	Adding new business logic as a CRT extension DLL .....	37
6.3	Adding a new RetailServer extension DLL.....	37
6.4	Adding new extension project to POS .....	37
7	Code implementation scenarios .....	38
7.1	Commerce Runtime extensibility scenarios.....	38
7.1.1	Extension property usage on CRT entities .....	38
7.1.2	Extension property usage on CRT entities with persistence.....	38
7.1.3	Extension property usage on CRT request and response types .....	39
7.1.4	Implementing a new CRT service that handles multiple different new requests.....	39
7.1.5	Implementing a new CRT service that handles a single new request.....	40
7.1.6	Implementing a new CRT service that overrides functionality of existing request .....	41
7.1.7	Implementing a new CRT entity and use it in new CRT service .....	41
7.1.8	Adding Pre- and Post-triggers for a specific request .....	42
7.2	RetailServer extensibility scenarios .....	43
7.2.1	Adding a new ODATA action to an existing controller .....	43
7.2.2	Adding a new simple controller for entity .....	44
7.3	RealTime Transaction service extensibility scenarios.....	45

7.3.1	Implementing a new Retail transaction service extension method in AX .....	45
7.3.2	How to call the new retail transition service method from CRT: .....	51
7.3.3	Persist Extension properties sent via existing RTS calls.....	52
7.4	Offline mode extensibility scenarios.....	52
7.4.1	How to call new/customized CRT service in offline:.....	52

## 1 Overview

Microsoft Dynamics AX for Retail provides mid-market and large retailers with a complete head office and point of sale (POS) solution that includes support for online and brick-and-mortar stores. It can help retailers increase financial returns, improve service, manage growth, reach customers, and streamline efficiencies.

The Retail SDK contains binaries, templates, sample source code, and deployment assets that you can use to customize the Retail system.

The new release of the Retail Sdk for AX7 is improved to previous versions. Its main goals are to provide more help to partners, ISVs and customers during their customization work. The two ways Microsoft is trying to accomplish this with are:

- By providing precise sample customizations shipped as part of a few sample projects in the Retail Sdk, and
- Give clear and extensive documentation how to accomplish the top extensibility scenarios.

The AX7 Retail Sdk provides the main means in order to customize the Dynamics AX Retail store side components. This document goes hand in hand with the Retail Sdk. It covers Retail Sdk for AX7 and its usage in depth. Additionally, many customization scenarios are highlighted here refer to exact sample extensions contained in the Retail Sdk, hence everyone with access to the Sdk has the sample code as well.

It is strongly advised you make yourself familiar with the overall Dynamics AX for Retail architecture and Retail Application Lifecycle Management, as the coverage of these is out this document's scope. Please refer to these links:

- Dynamics AX for Retail architectural overview: <http://go.microsoft.com/fwlink/?LinkId=724376>
- Retail ALM: <http://go.microsoft.com/fwlink/?LinkId=724377>.

## 2 Structure of this document

Overall the document is made up of two parts.

The first covers the Retail Sdk, build, configurations that can be done to be more productive. Additionally, typical branching strategy is discussed in order to work effectively in teams.

The second part contains details about the main extensibilities that are possible with the Retail Sdk. Anyone getting started with writing code against the Sdk can use it as a reference in order to get started.

### 2.1 Abbreviations

Through this document, some terms are used that may require more definition. Below is a list of them.

*Table 1 Abbreviations*

Abbreviation	Description
<b>CRT</b>	CommerceRuntime, the DLLs that include the business logic hosted by RetailServer, MPOS (in offline mode) and AX
<b>RS</b>	RetailServer. A host of the CRT. RetailServer is a layer on top of the CRT that adds authentication, authorization, exposed ODATA endpoints for entities and other functionality
<b>RTS</b>	Realtime Transaction service. An AX endpoint that allows for synchronous calls from CRT/RS to AX for real time data.
<b>CDX</b>	Commerce Data Exchange. Software components that provide syncing capability in both directions between AX and the store.
<b>Channel DB</b>	Channel or Store database.
<b>MEF</b>	Managed Extensibility Framework (MEF) is a component of <a href="#">.NET Framework 4.0</a> aiming to create lightweight, extensible applications. It aims to allow .NET application developers to discover and use extensions easily from existing code.
<b>ALM</b>	Application Lifecycle management. Product lifecycle management that includes source control, code integration, change management, versioning and many other aspects of product development.
<b>LCS</b>	Dynamics Lifecycle Services. Allows customers and partners to manage the application lifecycle.
<b>VSO</b>	Visual Studio Online. A hosted service that gives users ALM functionality.
<b>TFS</b>	Team Foundation Server. Similar to VSO.

## 3 Retail Sdk overview

The Retail Sdk includes binaries, code, code samples, templates, and tools that can be used to customize Microsoft Dynamics AX Retail functionality.

The Retail Sdk can be obtained in multiple ways. LCS is the main tool to deploy new environments, and the Retail Sdk is included in new development machines and hotfix packages.

On the high-level the Retail Sdk consists of many files, which are logically grouped in order to meet the main goals, described below.

### 3.1 Rapid development

This is the main focus; getting customizations written efficiently and correctly. The Retail Sdk allows the user to run applications directly on a single-computer demo environment by using Visual Studio's F5 functionality (run and debug). Most of the necessary "deployment chores" are done for the developer.

### 3.2 Full MSBuild integration

The Retail Sdk is a build system. A simple MSBuild command from the root of the Sdk allows to build everything. This eliminates guesswork about how and where to build from and guarantees consistency and reproducibility. Because of this, the Retail Sdk can be easily used in connection with any Application Lifecycle Management system (ALM), including Visual Studio Online. This includes automating the build.

Microsoft makes big efforts to ship the Retail Sdk in a consistent and buildable state. The goal is that it builds out-of-the-box with no additional steps (if software prerequisites are met).

### 3.3 Packaging tools

The Sdk includes tools that generate new packages that include all the things that are needed to deploy a service. As an example, if the CommerceRuntime is extended with a new custom service DLL, the Sdk will automatically include this new DLL in all appropriate packages (RetailServer, MPOSOffline). Or, if the database is being extended, the upgrade script is automatically included in both RetailServer and MPOSOffline packages as these are the ones that need to (potentially) run the channel database update.

Files that are shared exist only once in the Sdk. The packaging projects are setup in a way that they will pull in the right files for the package. That means that you only edit a `commerceruntime.config` file in one place. The same is the case for deployment-related script files, even though these need to be customized in only rare cases.

### 3.4 Facilitate better code separation

Whenever the Retail Sdk needs to be updated, a potential code merge is needed. This is true if existing code was changed. A few new features in the Retail implementation and the folder structure of the Sdk help to better separate customization code from sample code and therefore eliminate much of this problem. This is an ongoing process and more improvements in this area can be expected in the future.

### 3.5 Real-world implementation samples

In addition to the source code of some of the Retail implementation the Sdk also includes sample code that illustrates how certain scenarios should be implemented.

See section 4.5 for a detailed list of these samples.



## 4 Retail Sdk deep dive

### 4.1 Retail Sdk checklist

Below are the required and/or recommended steps to get started for development teams. More detailed steps are provided in the linked document paragraphs.

Table 2 Retail Sdk setup checklist for developing teams

Step	Reference	importance	Remarks
<b>Branching/source control</b>	0	Highly recommended	Support engineers may ask about what particular changes have been made. Without a source control system, getting help will be very hard.  Usually, this is a one-time setup step, carried out by one team member.
<b>Obtain the Retail Sdk</b>	4.2	Required	Needed if source control is not used
<b>Perquisites</b>	4.3	Required	
<b>Understand the Retail Sdk</b>	4.4, 4.6	Optional	These sections provide important knowledge about what the Retail Sdk contains (4.4) and how it works (4.6)
<b>Initial configuration</b>	4.8	Required	DLL naming, code signing, versioning
<b>Initial full build</b>	4.6	Optional	
<b>Samples</b>	4.5	Optional	The samples provide a good solutions to the most common extensibility problems
<i>At this point existing or sample solutions can be reviewed, executed and debugged</i>			
<b>Best practices</b>	6.1	Required	
<b>Start your work</b>	4.10, 4.11, 4.8	Required	Adding new projects (4.10), developer activities (4.11), new extension DLLs (4.8)
<b>Common customizations scenarios</b>	7	Optional	
<i>At this point your customized solution can be reviewed, executed and debugged</i>			

### 4.2 Obtaining the Retail Sdk

In case source control is used, check with your administrator. It is highly recommended to use source control.

In case source control is not used, the process is the same for major releases and cumulative updates. In these two cases, the full Sdk can be found on a single-box developer topology. In this case, it is immediately ready for configuration and use.

In the case of a hotfix, the Sdk is included in the hotfix package itself as a zipped folder. As usual with Retail (as opposed to AX X++ hotfixes) a hotfix is cumulative and includes all other fixes.

In either case (major, CU or hotfix) it is advised that you put the Sdk in a source control system like Visual Studio Online. See ‘

Application lifecycle management (ALM)’ for details. Once this is done, any team member can just “enlist” into the Retail Sdk (untouched, or already customized) by syncing to their source control system.

### 4.3 Prerequisites

In order to **code, build and run** components of the Retail Sdk plus your customizations, the following tools are required:

- For ModernPOS and CloudPOS: Visual Studio 2015 with Typescript 1.63 (LCS dev topology ok)
- For ModernPOS: Windows SDK for Windows 8.1 (<https://msdn.microsoft.com/en-us/windows/desktop/bg162891.aspx>)
- For StoreFront sample: ASP.NET MVC 4.0 (LCS dev topology ok)
- 150+ MB of available disk space (LCS dev topology ok)

This is a very short list of requirement, which allows developers to be productive on simple laptops. There is no prerequisite validation utility anymore.

In order to **execute** your customization, the normal prerequisites to run the Retail application apply. The suggested way to execute the customizations during development is on a single-box developer topology, either LCS cloud-hosted or downloaded.

### 4.4 Retail Sdk contents

The following folders and files are part of the Retail Sdk at the top level:

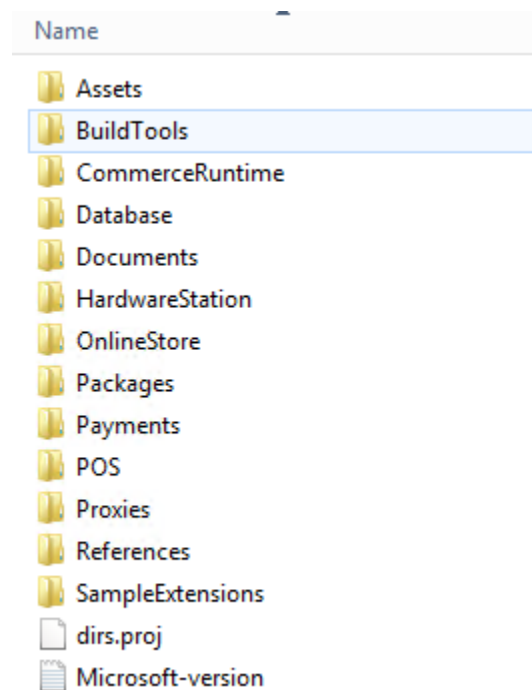


Figure 1 Retail Sdk structure

Folder/file	Description
<b>Assets</b>	Contains shared items like scripts and configuration files ( <b>commerceRuntime.config, dllhost.exe.config, etc.</b> ). The

	configuration files should be customized/edited right in this location. The projects that use them will pick them up accordingly.
<b>BuildTools</b>	<p>Contains anything related to the MSBuild and global configurations. <b>Customization.settings</b> is the main file that is used to setup the build system. Items this file controls and that customizers will likely change:</p> <ul style="list-style-type: none"> <li>- prefix for all built assemblies (AssemblyNamePrefix)</li> <li>- assembly version for all binaries (CustomAssemblyVersion)</li> <li>- file version for all binaries (CustomVersion)</li> <li>- name of the customization (CustomName)</li> <li>- description of the customization (CustomDescription)</li> <li>- publisher (CustomPublisher)</li> <li>- code signing (SignAssembly, AssemblyOriginatorKeyFile)</li> <li>- ModernPOS certificate path (ModernPOSPackageCertificateKeyFile)</li> <li>- files related to the customization (RetailServerLibraryPathForProxyGeneration, ISV_*)</li> </ul>
<b>CommerceRuntime</b>	Contains a Visual Studio solution file <b>CommerceRuntime.sln</b> and related C# projects (CommerceRuntime, PricingEngine)
<b>Database</b>	<p>Contains shared database scripts. The full script CommerceRuntimeScripts_Create.sql is only used to create a new base database. The Upgrade folder includes incremental scripts, from both Microsoft and the customizer.</p> <p>The existence of the database or the version in the database during a deployment controls which full or incremental scripts will be run. Only scripts that have not run before will be executed.</p>
<b>Documents</b>	Contains documentation for the Retail Sdk and samples
<b>HardwareStation</b>	Contains a Visual Studio solution file <b>HardwareStation.sln</b> and related C# projects (HardwareStation libraries and Webhost)
<b>OnlineStore</b>	Contains a Visual Studio solution file <b>OnlineStore.sln</b> and related C# projects (Ecommerce Sdk)
<b>Packages</b>	Contains multiple projects for package creation. These packages are going to be used to deploy via LCS. These packages also include the final <b>web.config</b> files.
<b>Payments</b>	Contains a Visual Studio solution file <b>PaymentSdk.sln</b> and related C# projects (PaymentSdk connectors, samples and functional test)
<b>POS</b>	<ul style="list-style-type: none"> <li>- <b>CloudPos.sln</b></li> <li>- <b>ModernPos.sln</b></li> <li>- folder Core: low-level shared POS code</li> <li>- folder ViewModels: shared POS view models</li> <li>- folder SharedApp: shared POS views</li> <li>- folder App: ModernPos-specific views and other items</li> <li>- folder Web: CloudPos-specific views and other items</li> </ul>
<b>Proxies</b>	Contains two Visual Studio projects that are referenced by others and contain interfaces and generated code that serves as a proxy client of RetailServer. Proxies.Retail.TypeScript is the TypeScript proxy, RetailProxy is the C# proxy. They are used by POS (both) and ECommerce Sdk.

<b>References</b>	This is the single place where all binaries live. The location is used to resolve any project's binary references. The list of files includes external non-Retail binaries as well as Microsoft's Retail binaries. Additionally, this directory serves as the global drop location for any binaries that is being built from the Retail Sdk.
<b>SampleExtensions</b>	Contains sample extensions
<b>dirs.proj</b>	Top-level MSBuild file that directs the build order
<b>Microsoft-version.txt</b>	File that includes Microsoft's version of the Retail Sdk. Do not edit this file.

The C# source code in the Retail Sdk uses the 'Contoso' namespace. This makes it easier to distinguish between Microsoft's and your own types, as Microsoft uses 'Microsoft.Dynamics'. See Figure 2 for an illustration.

If you are referencing a type from Microsoft's binary, you would reference it with Microsoft.Dynamics and can be sure it is not from the Retail Sdk but from a referenced binary.

```

1 namespace Contoso
2 {
3     namespace Retail.Ecommerce.Publishing
4     {
5         using System;
6         using System.Collections.Generic;
7         using System.Diagnostics;
8         using System.Linq;
9         using System.Text;
10        using Microsoft.Dynamics.Commerce.Runtime.DataModel;
11        using Retail.Ecommerce.Sdk.Core.Publishing;
12
13        internal class ChannelPublisher : IChannelPublisher
14        {
15            /// <summary>

```

Figure 2 C# namespace

## 4.5 Retail Sdk Extensibility samples and tools

The RetailSdk includes extensibility samples. These samples are a good way to get started learning about different ways to customize Retail.

Additionally, these projects can also be used as boilerplate projects to get specific customizations started. For example, if a new CommerceRuntime extension is started, the simplest way is to copy one of the CommerceRuntime sample projects into a new folder, for example RetailSdk\Extensions, adjust the project import paths and then start working in it (see 4.10.1).

Further step-by-step instructions are available in the Retail Sdk\Documents folder for most of the samples. Below is a high-level list of the samples.

Table 3 List of sample extensions part of the shipped Retail Sdk

Sample name	Details
-------------	---------

<b>CrossLoyalty</b>	<p>Consider that there are 2 retailers, AdventureWorks and Contoso. As a part of a deal, Contoso retailer will accept loyalty points of AdventureWorks.</p> <p>The sample shows how to create a simple new CRT service and call it once a button in MPOS is clicked. It simulates the cross loyalty scenario.</p> <p>Changes are in AX configuration, CRT, RetailServer, RetailProxy and Point of Sale (both Modern POS and Cloud POS). Offline mode for Modern POS is supported for this sample.</p>
<b>EmailPreference</b>	<p>The sample shows the usage of extension properties to extend an entity. Entity is extended in AX, persisted in both AX and Channel databases, and POS UI allows to access the value. Additionally, the new value is written via the RetailRealtimeTransaction service synchronously to AX. No customizations are needed in the areas of CommerceRuntime or RetailServer (Extension properties flow automatically).</p> <p>Changes are in AX forms, AX tables, AX RTS client, CDX, Channel DB, Point of Sale (both Modern POS and Cloud POS). Offline mode is not supported for this sample.</p>
<b>StoreHours</b>	<p>The sample shows how to create a new business entity (StoreHours) accross both AX and the channel.</p> <p>Changes are in AX tables, CDX, Channel DB, CRT, RetailServer, Point of Sale (both Modern POS and Cloud POS). Offline mode for Modern POS is supported for this sample.</p>
<b>HealthCheck</b>	<p>The sample shows how to expand an existing CRT service with additional functionality. In this case, some other system's health could be checked as part of the already existing RunHealthCheckServiceRequest.</p> <p>Changes are in CRT and CRT Test Host.</p>
<b>ExtensionProperties</b>	<p>The sample shows these customization strategies for the CommerceRuntime: Extension property for service, entity, request, and response. Triggers, Notifications and Notification Handlers.</p>
<b>CommerceRuntime Test Host</b>	<p>This tool mimics a typical CRT host (similarly to RetailServer) and allows for simple testing of CRT extensions without requiring changes in RetailServer or UI clients. Just register the services required for the CRT, and run it.</p> <p>Note: RealTimeTransaction service calls that may be part of a CRT extension will not function properly. In order to test these, use the "RetailServer Test Client" instead.</p>
<b>RetailServer Test Client</b>	<p>This simple application is very useful to make either 1) RetailServer calls or 2) offline mode calls through the RetailProxy or 3) both. It acts as a client, similarly as the POS clients, but requires no UI changes and therefore allows for more rapid testing and</p>

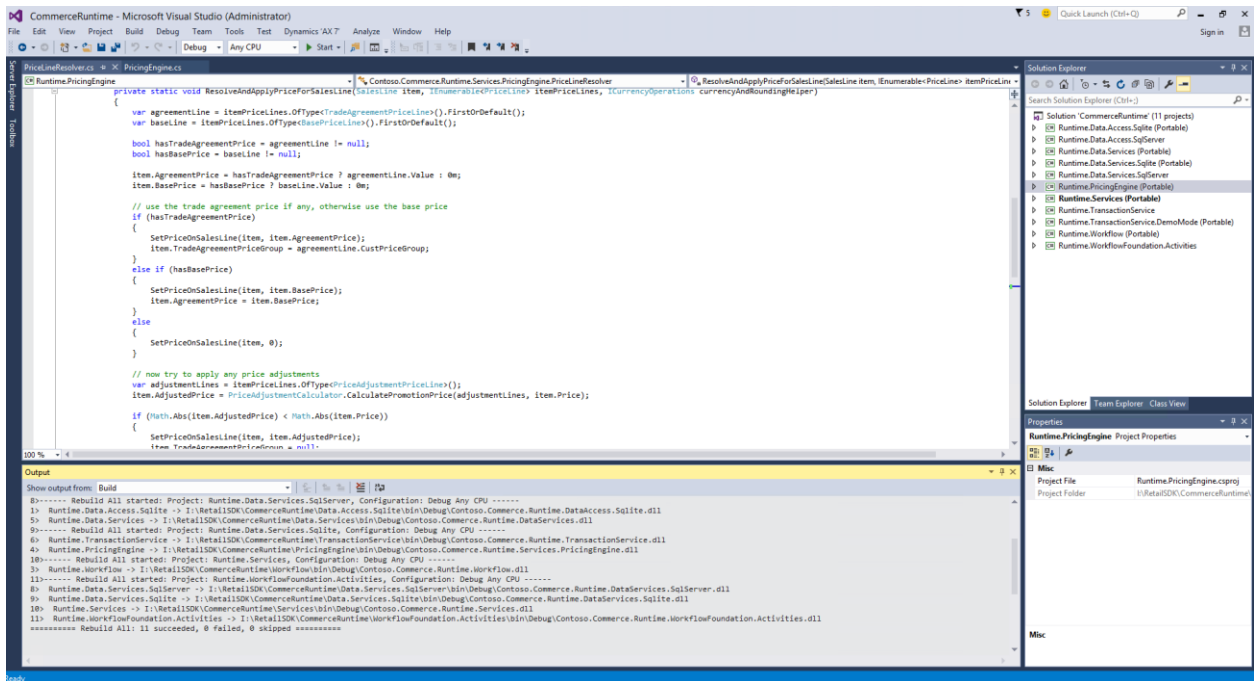
	development. Use this tool to verify customizations in the areas of Channel database, RTS, CRT, and/or RetailServer before you hand them off to the UI team.
<b>OnlineStore</b>	An ASP.NET implementation that showcases the use of the Ecommerce SDK. It includes both a store front and a publishing job.
<b>OpenIdConnectUtility</b>	Allows to call RetailServer in a customer context (C2) with a Google Identity.
<b>CashDispenser (HardwareStation)</b>	Shows how to customize HardwareStation so a cash dispenser can be integrated
<b>Rambler (HardwareStation)</b>	Shows how to customize HardwareStation and how to integrate a Rambler Mobile Card reader

#### 4.6 Dependencies, build order and full build

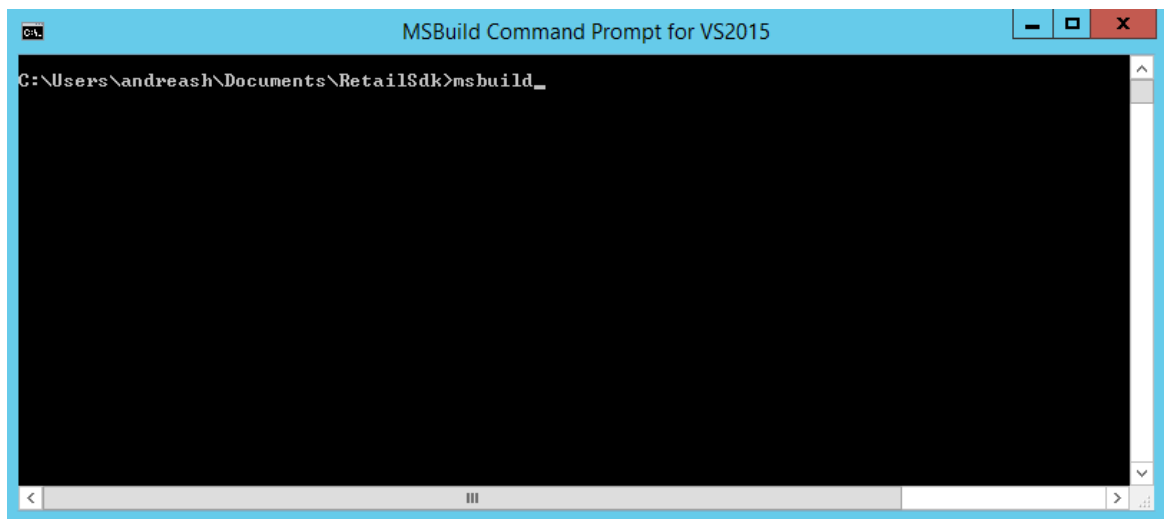
The Figure 3 shows a high-level logical dependency tree within the Retail Sdk. It does not show the references to all Microsoft files or assets. These can be seen by looking at the Visual Studio project and solution files in more detail.

A couple of important things should be highlighted:

- The RetailServer API is consumed by a few projects by means of automatically generated client proxy code. This allows for more rapid development and reduces opportunities for errors and bugs. By default, the Retail Sdk uses Microsoft's official DLL to generate the client code. A customizer can switch to their own DLL (in Customization.settings), and therefore automatically generate the proxy code for their customized RetailServer API. After switching the DLL, a developer may need to change some implements inside the RetailProxy project. The reason for this is that ModernPOS in offline mode needs to talk directly to the CommerceRuntime and that code needs to be implemented. But, there is no guessing here, the C# compiler will force it.
- The packaging projects generate the deployment packages in the way LCS expects them. By default, these will only ship the Microsoft assets (un-customized) plus the proxy DLLs. Anything else that should be included must be explicitly named in Customization.settings. This is done on purpose. It reduces the deployed custom code and allows for binary patches. Example: A customization adds a new CommerceRuntime service, a new RetailServer controller. In that case 2 new DLLs would be registered to be included in the packages, and they are automatically included up in all relevant places. The packages will NOT have all re-compiled binaries from the Sdk.
- There is not a single Visual Studio solution that includes all projects. Because very little couplings between the different Visual Studio projects, you can open multiple projects or solutions side-by-side and compile the appropriate one after a change.



- Even if you did not customize every component, it is easiest to get the final deployment packages by building the whole Retail Sdk. To do that, open a “MSBuild Command Prompt for VS2015” window and type `msbuild /t:Rebuild` (or `msbuild /p:Configuration=Release` for a non-debug version):



This command will build all projects. This is also a great way to check and make sure there are no implementation or code bugs. If there are, the build will fail and indicate what failed (similarly what Visual Studio would show):

```

MSBuild Command Prompt for VS2015

"C:\Users\andreash\Documents\RetailSdk\Pos\App\Pos.App.jsproj.metaproj" <default target> <54> ->
"C:\Users\andreash\Documents\RetailSdk\Pos\App\Pos.App.jsproj" <default target> <58> ->
<SplitResourcesPri target> ->
  MakePRI : warning 0xdef01051: No default or neutral resource given for 'resources/string_189'.
  row an exception for certain user configurations when retrieving the resources. [C:\Users\andreash\
  Pos\App\Pos.App.jsproj]

"C:\Users\andreash\Documents\RetailSdk\dirs.proj" <default target> <1> ->
"C:\Users\andreash\Documents\RetailSdk\SampleExtensions\dirs.proj" <Build target> <36> ->
"C:\Users\andreash\Documents\RetailSdk\SampleExtensions\CommerceRuntime\dirs.proj" <Build target>
"C:\Users\andreash\Documents\RetailSdk\SampleExtensions\CommerceRuntime\Extensions.TestHost\RunTi
.csproj" <Build target> <38> ->
<CoreCompile target> ->
  Program.cs(51,17): error CS1003: Syntax error, ',' expected [C:\Users\andreash\Documents\Retail
  ommerceRuntime\Extensions.TestHost\Runtime.Extensions.TestHost.csproj]
  Program.cs(51,18): error CS1002: ; expected [C:\Users\andreash\Documents\RetailSdk\SampleExtens
  xtensions.TestHost\Runtime.Extensions.TestHost.csproj]

  5 Warning(s)
  2 Error(s)

Time Elapsed 00:01:21.21
C:\Users\andreash\Documents\RetailSdk>

```

For detailed help on MSBuild, please see <https://msdn.microsoft.com/en-us/library/0k6kkbsd.aspx>.

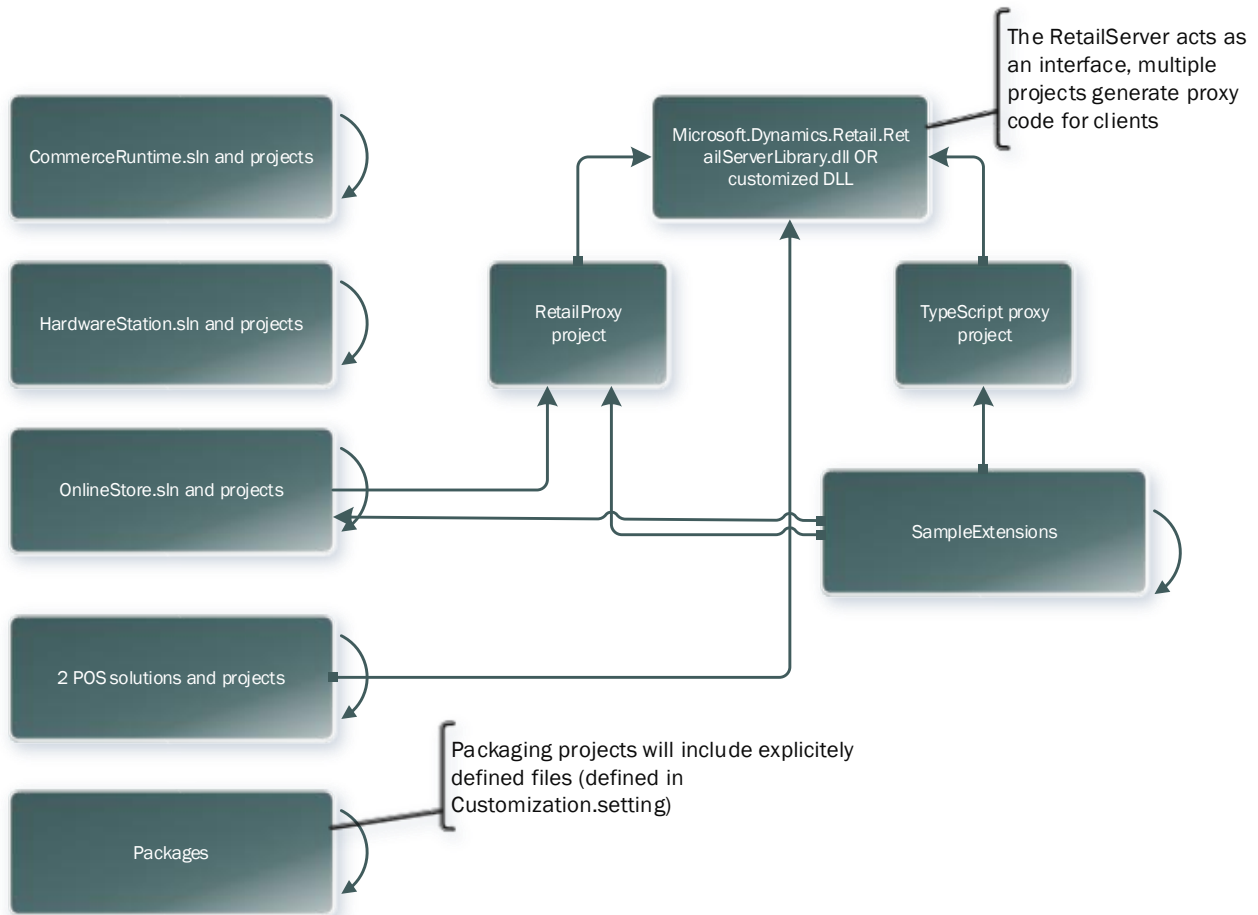
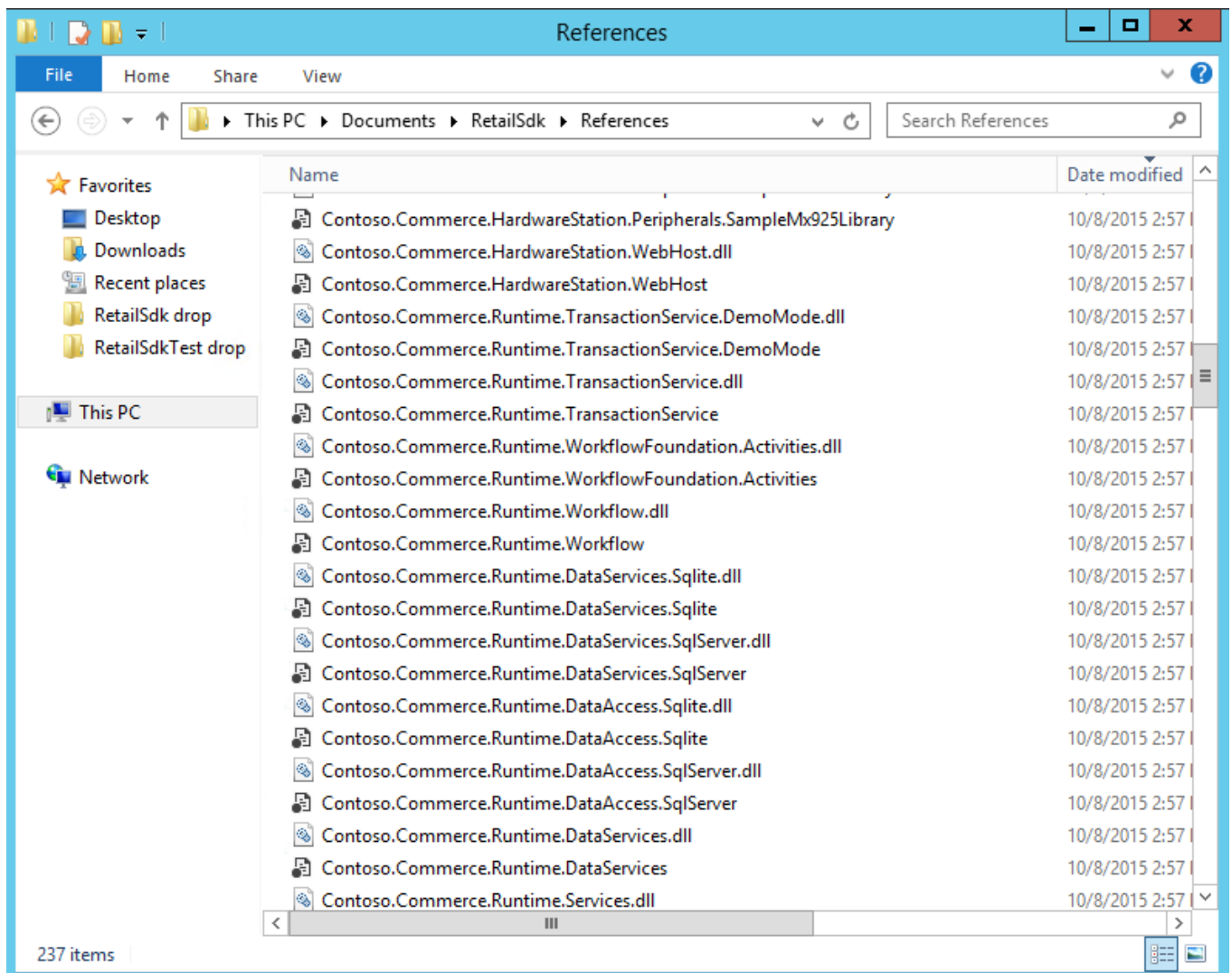


Figure 3 Retail Sdk logical project dependencies



The binaries created by the build are automatically copied to the Sdk's References folder. The References folder also includes all the other binaries. Notice, that there are no DLLs overwritten as they are all prefixed with a name you can define in Customization.settings (here 'Contoso'):



#### 4.7 Minimal configuration

Do you just quickly want to build the Retail Sdk, or run POS in the debugger on a demo machine?

The good thing is, you have everything to build individual solutions, projects or the whole Retail Sdk (with MSBuild).

#### 4.8 Normal configuration/code signing

In order to do work that goes beyond just running things, it is recommended to do some configurational changes first.

For ModernPOS only: You should create your own certificate for app package signing. This certificate needs to be created unless you do not already have one. Follow these instructions to create a pfx file:

[https://msdn.microsoft.com/en-us/library/windows/desktop/jj835832\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/jj835832(v=vs.85).aspx)

Then copy the PFX file to the BuildTools folder, and update the BuildTools\Customization.settings file with the correct name (ModernPOSPackageCertificateKeyFile).

BuildTools\Customization.settings is the location of most of the configuration values for the Sdk. The highlighted items in Figure 4 show the global values. These control how built binaries, components and packages are being named, versioned and code-signed.

```
<!-- This section is for global settings and code signing. Any built file will inherit these values if applicable.
also use these values during package generation. -->
<AssemblyNamePrefix>MyCompany</AssemblyNamePrefix>
<CustomAssemblyVersion Condition="'$(CustomAssemblyVersion)' == ''>1.0.0.0</CustomAssemblyVersion>
<CustomVersion Condition="'$(CustomVersion)' == ''>1.0.0.1</CustomVersion>
<CustomName Condition="'$(CustomName)' == ''>MyCompany Retail Customization</CustomName>
<CustomDescription Condition="'$(CustomDescription)' == ''>MyCompany Retail Customization</CustomDescription>
<CustomPublisher Condition="'$(CustomPublisher)' == ''>MyCompany Ltd.</CustomPublisher>
<CustomCopyright Condition="'$(CustomCopyright)' == ''>Copyright © 2015</CustomCopyright>

<SignAssembly Condition="'$(SignAssembly)' == ''>true</SignAssembly>
<DelaySign Condition="'$(DelaySign)' == ''>>false</DelaySign>
<AssemblyOriginatorKeyFile Condition="'$(AssemblyOriginatorKeyFile)' == '' and '$(SignAssembly)' == 'true'>
$(MSBuildThisFileDirectory)\StrongNameSigningCert-Contoso.snk</AssemblyOriginatorKeyFile>

<ModernPOSPackageCertificateKeyFile Condition="'$(ModernPOSPackageCertificateKeyFile)' == ''>
$(MSBuildThisFileDirectory)\ModernPOSApexSigningCert-Contoso.pfx</ModernPOSPackageCertificateKeyFile>

<RetailServerLibraryPathForProxyGeneration Condition="'$(RetailServerLibraryPathForProxyGeneration)' == ''>
$(SdkReferencesPath)\Microsoft.Dynamics.Retail.RetailServerLibrary.dll</RetailServerLibraryPathForProxyGeneration>
</PropertyGroup>
```

Figure 4 Global configuration values for the Sdk

It is good practice to sign your assemblies with a strong name, even though it is not required. Please refer to [https://msdn.microsoft.com/en-us/library/6f05ezxy\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/6f05ezxy(v=vs.110).aspx) to learn how to create your own key file if you do not have one yet.

Both the strong name key file and the app package signing certificate could be stored inside the BuildTools folder.

The property RetailServerLibraryPathForProxyGeneration can be used to set a different RetailServer dll for proxy generation. You will only need this after you changed the RetailServer API and you need clients to call the new API.

Customization.settings is also the place to define your new customization assets, like binaries, config files, SQL update scripts etc.

```
<!-- This section is for additional files related to a customization. -->
<ItemGroup>
  <!-- This is where any additional CRT dlls should be specified -->
  <!--ISV_CommerceRuntime_CustomizableFile Include="$(SdkReferencesPath)\MyCrtExtension.dll" /-->

  <!-- This is where any additional RetailServer dlls should be specified -->
  <!--ISV_RetailServer_CustomizableFile Include="$(SdkReferencesPath)\MyRetailServerExtension.dll" /-->

  <!-- This is where any additional hardware station files should be specified -->
  <!--ISV_HardwareStation_CustomizableFile Include="$(SdkReferencesPath)\MyHardwareStationExtension.dll" /-->

  <!-- This is where any additional db upgrade scripts should be specified -->
  <!-- ISV_CustomDatabaseFile_Upgrade_Custom Include="$(SdkRootPath)\Database\Upgrade\Custom\SqlUpdatev1.sql
</ItemGroup>
```

Figure 5 File-related MSBuild items

## 4.9 Adding a payment connector

### 4.9.1 Adding the files

The deployment projects are setup to include the correct payment binaries and web files into the final deployment packages. The only thing required by the user of the Retail Sdk is to add the actual files into the assumed locations of the Sdk.

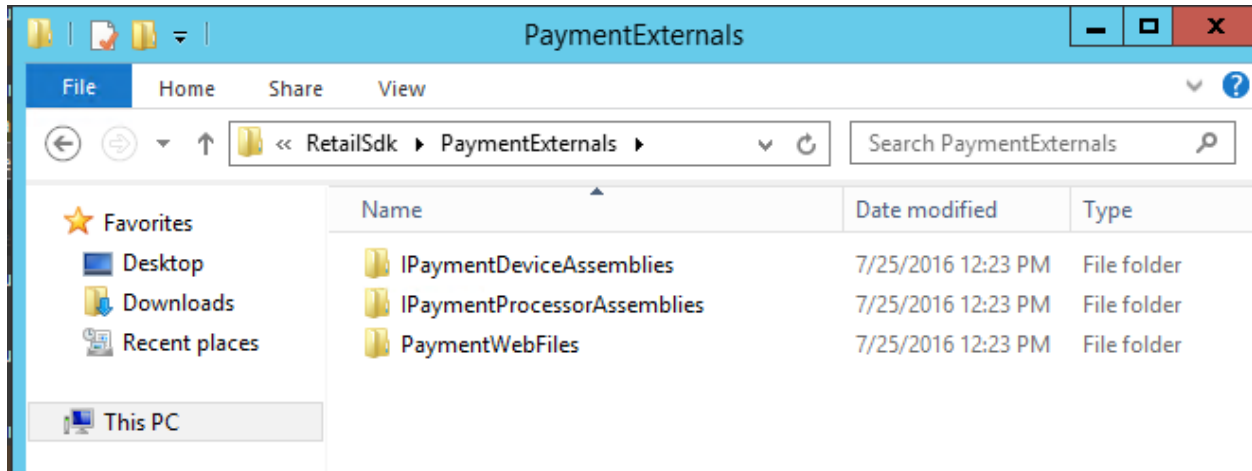


Figure 6 Location of the 3 payment-related folders in the Retail Sdk

The files are usually provided by a 3<sup>rd</sup> party vendor. As soon as they are added into the expected 3 folders, the packages will include them and the deployment will install them.

### 4.9.2 Register payment accepting page URL

In addition to file dropping, you also need to modify some files if you plan to use payment accepting page in Cloud POS or Modern POS.

A payment accepting page is an embedded page from an external website to accept payment card information. It is used in AX, Ecommerce, Cloud POS and Modern POS. This page is hosted on an external server with a different domain. Both Cloud POS and Modern POS only allow to open a page from pre-registered URLs, basically a whitelist. For example, your payment accepting page URL is <https://www.mypaymentsite.com/paymentaccept/cardpage.aspx>. You need to add the URL (or part of the URL) to the following places.

For Cloud POS:

1. Edit RetailSDK\POS\Web\web.config.
2. Find the line: <add name="Content-Security-Policy"...
3. Following "default-src", add payment URL origin to the value e.g. "<https://www.mypaymentsite.com/>".
4. Save the file permanently to your source control.

For Modern POS:

1. Edit RetailSDK\POS\App\package.appxmanifest file.
2. Find <ApplicationContentUriRules> node.
3. Add a Rule like this: <Rule Match="<https://www.mypaymentsite.com/>" Type="include" />.

4. Save the file permanently to your source control.

**Note**, if there are multiple URLs for different environments, you can add all of them to Cloud POS and MPOS similarly.

#### 4.9.3 Register payment device assembly

During file dropping, if you copied any IPaymentDevice Assemblies, e.g. MyPaymentDevice.dll, you need to register the payment device assembly in the config file of hardware station.

For remote hardware station (IIS)

1. Edit RetailSDK\Packages\HardwareStation\web.config file
2. Find <composition> node under <hardwareStation> node.
3. Under <composition> node, add a new line like this: <add source="assembly" value="MyPaymentDevice" />.
4. Save the file permanently to your source control.

For local hardware station (IPC)

1. Edit RetailSDK\Assets\dllhost.exe.config file
2. Find <composition> node under <hardwareStation> node.
3. Under <composition> node, add a new line like this: <add source="assembly" value="MyPaymentDevice" />.
4. Save the file permanently to your source control.

**Note**, you only need to register the assembly which implements the IPaymentDevice interface. The assembly name does not include “.dll”.

### 4.10 Customizing the build

#### 4.10.1 Adding new projects

Adding new projects to the Sdk’s build system is simple. You can either clone one of the many existing projects or start a new project. Just make sure to make some adjustments with a text editor, as illustrated in Figure 7 (relative path of the Imports should be adjusted, AssemblyName should use the predefined Property “AssemblyNamePrefix”). This is required in order to get versioning, code signing, uniform assembly naming, automatic drop to the References folder and other tasks for free.

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <Project ToolsVersion="14.0" DefaultTargets="Build" xmlns="http://schemas.microsoft.com/d
3  <Import Project="..\..\..\BuildTools\Microsoft.Dynamics.RetailSdk.Build.props" />
4  <Import Project="$(MSBuildExtensionsPath)\$(MSBuildToolsVersion)\Microsoft.Common.props"
   '$(MicrosoftCommonPropsHasBeenImported)' != 'true' />
5  <Import Project="..\..\..\BuildTools\Microsoft.Dynamics.RetailSdk.Build.settings" />
6  <PropertyGroup>
7    <Configuration Condition=" '$(Configuration)' == '' ">Debug</Configuration>
8    <Platform Condition=" '$(Platform)' == '' ">AnyCPU</Platform>
9    <ProjectGuid>{2CB843E4-8BEB-4A57-A72B-66D1849FFED6}</ProjectGuid>
10   <OutputType>Exe</OutputType>
11   <RootNamespace>Contoso.Retail.Ecommerce.Publishing</RootNamespace>
12   <AssemblyName>$(AssemblyNamePrefix).Retail.Ecommerce.Publishing</AssemblyName>
13   <ShippingSourceCode>true</ShippingSourceCode>
14   <TargetFrameworkProfile />
15   <TargetFrameworkVersion>v4.5.1</TargetFrameworkVersion>
16   <DefaultLanguage>en-US</DefaultLanguage>
17 </PropertyGroup>
18 <PropertyGroup Condition=" '$(Configuration)|$(Platform)' == 'Debug|AnyCPU' ">
27 <PropertyGroup Condition=" '$(Configuration)|$(Platform)' == 'Release|AnyCPU' ">
35 <ItemGroup>
11 <ItemGroup>
16 <ItemGroup>
22 <ItemGroup>
39 <ItemGroup>
49 <Import Project="$(MSBuildToolsPath)\Microsoft.CSharp.targets" Condition=" '$(WindowsAppl
50 <Import Project="$(SdkRootPath)\BuildTools\Microsoft.Dynamics.RetailSdk.Build.targets" /
51 </Project>

```

Figure 7 Sdk's project file template

The following list is a step-by-step guide to accomplish adding a new project:

Table 4 Steps to add a new DLL project

### Step

1. Create a new folder called "Extensions" under the Retail Sdk root
2. Copy a sample project from SampleExtensions to Extensions folder for reuse

For CRT extension: copy Retail

Sdk\SampleExtensions\CommerceRuntime\Extensions.StoreHoursSample folder to Retail  
Sdk\Extensions\YourCRTEExtensionName folder

For RS extension: copy Retail Sdk\SampleExtensions\RetailServer\Extensions.StoreHoursSample  
folder to Retail Sdk\Extensions\YourRSEExtensionName folder

3. rename the csproj file to a name of your choice
4. Using a text editor, edit the csproj file so that all <Import> directives point to the correct relative paths
5. Using a text editor, edit the csproj file's assemblyname property to something like:  
"<AssemblyName>\$(AssemblyNamePrefix).CRTEExtensionName</AssemblyName>". Use the  
\$(AssemblyNamePrefix) property.
6. Using a text editor, add a new entry in Retail Sdk\dirs.proj so that the new project or project  
structure is included in the build
7. (Optional) Create a Visual Studio solution with the projects you care about for simpler  
development

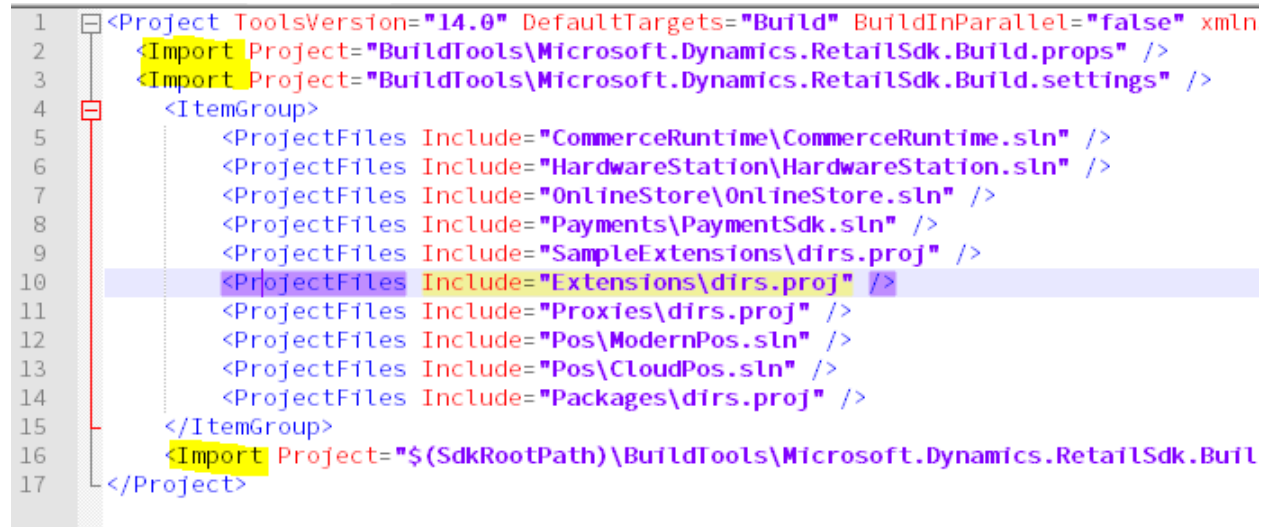
8. Run “msbuild /t:Rebuild” from the root of the Retail Sdk and verify that the DLL from this new extension project is dropped correctly into the References folder

9. Update Customization.settings to include this new DLL in the appropriate ISV\_\* section

10. Update commerceRuntime.config (for CRT DLLs) or web.config in Packages\RetailServer (for RS DLLs)

#### 4.10.2 Changing build order or adding to the build

The whole directory tree of the Retail Sdk is built with the help of MSBuild traversal files (dirs.proj).



```
1 <Project ToolsVersion="14.0" DefaultTargets="Build" BuildInParallel="false" xmlns="http://schemas.microsoft.com/build/2009"
2   <Import Project="BuildTools\Microsoft.Dynamics.RetailSdk.Build.props" />
3   <Import Project="BuildTools\Microsoft.Dynamics.RetailSdk.Build.settings" />
4   <ItemGroup>
5     <ProjectFiles Include="CommerceRuntime\CommerceRuntime.sln" />
6     <ProjectFiles Include="HardwareStation\HardwareStation.sln" />
7     <ProjectFiles Include="OnlineStore\OnlineStore.sln" />
8     <ProjectFiles Include="Payments\PaymentSdk.sln" />
9     <ProjectFiles Include="SampleExtensions\dirs.proj" />
10    <ProjectFiles Include="Extensions\dirs.proj" />
11    <ProjectFiles Include="Proxies\dirs.proj" />
12    <ProjectFiles Include="Pos\ModernPos.sln" />
13    <ProjectFiles Include="Pos\CloudPos.sln" />
14    <ProjectFiles Include="Packages\dirs.proj" />
15  </ItemGroup>
16  <Import Project="$(SdkRootPath)\BuildTools\Microsoft.Dynamics.RetailSdk.Build.targets" />
17 </Project>
```

Figure 8 Top-level MSBuild traversal file

Figure 8 shows the main traversal file of the Retail Sdk. Similar files may exist in sub directories as well. Notice that Visual Studio solution files (sln files) are very similar to traversal files. Both “direct” the MSBuild engine to process other build scripts.

Once new code will be added, most of that should live in a new folder (see details in Code implementation) and would need to be added to the traversal structure by adding to one or multiple dirs.proj file. The highlighted “Extensions” folder in Figure 8 shows an example (line 10). The quickest way to get started with a new dirs.proj file would be to copy an existing one, correct the Import paths and update the ProjectFiles ItemGroup.

#### 4.10.3 Build script customization

When you need to implement some new build steps, keep in mind that the existing scripts may be updated by a Retail Sdk update later on. Best practice is to minimize the editing of any file or add new ones instead.

If you need new global MSBuild properties, BuildTools\Microsoft.Dynamics.RetailSdk.Build.props would be a good place to add them. Likewise, BuildTools\Microsoft.Dynamics.RetailSdk.Build.targets could be used to add new build processing targets.

If there is only one project needing some special handling, it would be better to explicitly do the change there. If you need new local MSBuild properties, add a new file called local.props in the same directory. Or, add a local.targets if you need some local build processing targets.

#### 4.11 Developer activities (update, run, debug, etc.)

On a developer topology computer, there is no “real” deployment needed, as the base versions of the different components are already pre-installed, in most cases. Also, working on a customization and updating the current developer machine does not require a re-deployment. However, it is suggested to follow a few steps that are simple and also have the benefits to roll back very quickly to compare different versions.

##### 4.11.1 One-time RetailServer setup for rapid development

RetailServer is an IIS web application. As mentioned above, it is advantageous to have a backup to fall back to. Additionally, to avoid any SSL certificate errors, it is best to re-use the same web application and its certificate mapping and hostname settings (do not use localhost in the url). The following steps take that into account:

*Table 5 Steps for RetailServer re-deployment on developer machine*

Step
<b>1. Find the physical path for RetailServer using IIS management console</b>
<b>2. Make a copy of the folder, indicate “RetailSdk” in the name of the new folder. For any further steps, only make changes in the new folder</b>
<b>3. Using IIS management console, “remap” the web application to use the new RetailSever folder created during step 2.</b>
<b>4. Enable RetailServerDropBinaryOnBuild and configure path (see below)</b>
<b>5. (automatic if step 4. was done) On compilation of CRT or RetailServer extension DLLs they are being dropped to the new RetailServer folder</b>
<b>6. Web.config file: Update the composition section with your new RetailServer dll (if you are extending RetailSever)</b>
<b>7. commerceRuntime.config: Update the composition section with your CRT extension dll</b>
<b>8. At this point, a simple re-compilation of the extension DLLs will automatically dropped</b>

Regarding step 4: As shown in Figure 3, there are a lot of things that are dependent of the RetailServer interface. It is likely someone would change it. On a developer topology machine, someone may want to immediately try out a change. That means that any CommerceRuntime and RetailServer extension DLLs would have to be copied into the bin folder of the locally installed RetailServer web application. This can be automated.

To set this up, a user should create a global MSBuild settings file at “Retail SDK \BuildTools\globaluser.props” that is user-specific (should not be added the source control). With this content (copy most from Customization.settings):



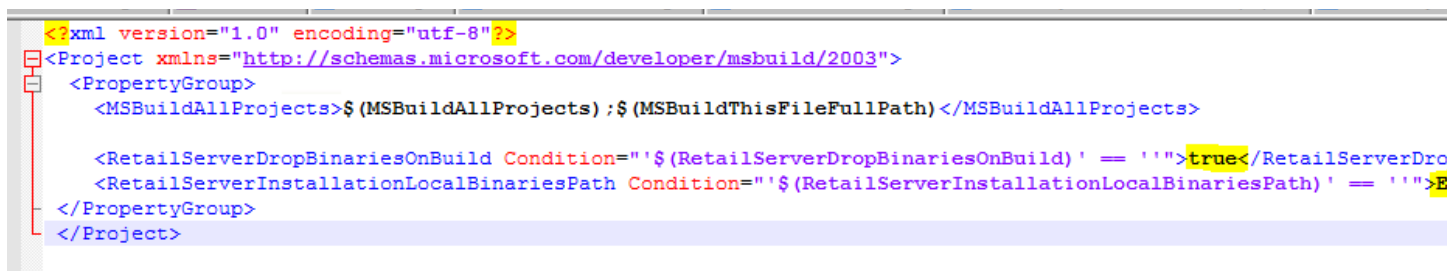


Figure 9 Enabling RetailServer binary drop for rapid development

Note: Do not run the UpdateRetailServer.ps1 deployment script on a developer machine. It is only designed to be run via an official LCS deployment and could cause issues.

It is noted here, that the same mechanism for rapid deployment on a development machine is available for HardwareStation. It gets configured in a similar way.

4.11.2 Debugging RetailServer and CRT code

The following steps are for developers that want to exercise a use case with a POS client, but want to step through the RetailServer or CRT customized code. Reasons could be:

- developer has verified with Fiddler that the calls are being issued but the response indicates that an error occurred, or
- the Event Viewer shows that an error occurred.

Table 6 How to debug RetailServer

Steps
1. Attach Visual Studio Debugger to the w3wp.exe process for RetailServer (or all if you are unsure which one it is)
2. Open the source code you would like to debug and set a breakpoint
3. Execute your client code (ModernPOS, CloudPOS, RetailServer TestClient, Fiddler HTTP replay, etc.)

Note: RetailServer traces can be inspected locally in the Event Viewer at “Event Viewer (Local)/Applications and Services Logs/Microsoft/Dynamics/Commerce-RetailServer/Operational”.

4.11.3 Debugging ModernPOS with F5 from Retail Sdk

These steps are based on a developer topology machine that has the official ModernPOS already installed. If this is not the case, ModernPos requires more steps to run from Visual Studio (ClientBroker and certificate-related). It is easiest to install ModernPOS once.

Table 7 Steps to debug Modern POS

Step
1. In Visual Studio, open the ModernPOS.sln of the Sdk
2. Re-build the solution and run it via F5.
3. You may be asked to uninstall the already installed ModernPOS, enable UAC, reboot, etc.). Carry out the steps and hit F5 again



4. Activate the device and login (use IIS management console to inquire for the url, use 000160 and 123 if you use Contoso demo data)
5. Set breakpoints at any TypeScript or JavaScript file

Note: ModernPOS traces can be inspected locally in the Event Viewer at “Event Viewer (Local)/Applications and Services Logs/Microsoft/Dynamics/Commerce-ModernPOS/Operational”.

#### 4.11.4 Debugging CloudPos with F5 from Retail Sdk

These steps are based on a developer topology machine that has the official CloudPos already installed. If this is not the case, or if you want a different url please adjust the steps.

Table 8 Steps to debug CloudPOS

##### Steps

1. In IIS management console/CloudPOS/Advanced Settings.../physical path, change the path to point to the Web folder of the Retail Sdk

**Note:** If you ever want to revert back, simply change the physical path back to the original setting. This will essentially roll CloudPOS back to the installed version.

2. In Visual Studio, open the CloudPOS.sln solution from the Sdk
3. Change the Pos.Web’s project properties to use IIS, and browse to the same url as CloudPOS was already installed at (use IIS management console to find it):

The screenshot shows the 'Properties' window for the 'Pos.Web' project in Visual Studio. The 'Web' tab is selected. Under 'Start Action', the 'Start URL' option is selected with the value 'https://usnconeboxax1pos.cloud.onebox.dynamics.com/'. Under 'Servers', the 'Apply server settings to all users (store in project file)' checkbox is checked, and the 'Local IIS' server is configured with the 'Project Url' set to 'https://usnconeboxax1pos.cloud.onebox.dynamics.com/'.

4. Delete all browsing history in Internet Explorer and close all your Internet Explorer instances
5. Hit F6 to launch CloudPOS in the debugger, and debug any TypeScript or Javascript file

**Note: If you get an access denied error, you may need to allow IIS to read the files from this folder. Add Read & Execute, list folder contents, and Read permissions to the local Windows group called “Users”.**

Note: CloudPOS traces can be inspected locally in the Event Viewer at “Event Viewer (Local)/Applications and Services Logs/Microsoft/Dynamics/Commerce-CloudPOS/Operational”.

#### 4.11.5 Verifying deployment and customizations of ModernPOS in offline mode

It is relatively easy to test customization in online mode. This can be accomplished by just debugging ModernPOS and hitting the correct RetailServer; both the client and server code can be tested or debugged. Offline mode is important to test explicitly as missing deployment and configuration issues can only be caught at runtime and because some of the code paths are different.

Below are the steps that can be carried out on any machine in a multi-box deployment or on the development topology machine (Note that you cannot access a single-box environment using the storage emulator from a different machine and test the offline mode. This is not supported).

*Table 9 ModernPOS offline deployment verification steps*

##### Steps

**1. Make sure both RetailSdk\Assets\commerceruntime.config and RetailSdk\Assets\CommerceRuntime.MPOSOffline.config have been updated with the equivalent changes**

**2. Make sure dllhost.exe.config specifies the assembly and type for the RetailProxy correctly:**  
`<add key="RetailProxyAssemblyName" value="Contoso.Commerce.RetailProxy" />`  
`<add key="AdaptorCallerFullTypeName"`  
`value="Contoso.Commerce.RetailProxy.Adapters.AdaptorCaller" />`

This is needed so that ClientBroker can launch the correct DLL via reflection.

**3. Ensure RetailSdk\Buildtools\Customization.settings lists all customization files correctly (ISV\_CommerceRuntime\_CustomizableFile, ISV\_RetailServer\_CustomizableFile, ISV\_HardwareStation\_CustomizableFile, ISV\_CustomDatabaseFile\_Upgrade\_Custom). If dlls are missing runtime errors will occur. If a database file is missing the offline database will be missing SQL objects which will also cause runtime errors.**

**4. From the root of the RetailSdk, type “msbuild /t:Rebuild”. This will build all binaries, the ModernPOS application, and create a new package for ModernPOSOffline (apart from all the others).**

**5. Make sure ModernPOS Appx package is un-installed. Use the following PowerShell command (run as administrator):**

```
Get-AppxPackage | ? {$_.Name -eq 'Microsoft.Dynamics.Retail.Pos'} | Remove-AppxPackage
```

**6. Use “Programs and Features” and ensure that “Microsoft.Dynamics.Retail.Pos” is uninstalled**

**7. If you are using a test certificate as the ModernPOSPackageCertificateKeyFile (that is the case by default), you need to make this certificate trusted. Double click the pfx file specified as ModernPOSPackageCertificateKeyFile in RetailSdk\Buildtools\Customization.settings in Windows Explorer, and import it into the store for Trusted Root Certification Authorities on the local machine. The default password is an empty string.**

**8. Find and run the newly built installer. It will be dropped into the References folder of the Retail Sdk.**

---

Contoso.ModernPOSSetupOffline.exe (change the assembly prefix accordingly if you changed it for your environment).

This will install and configure ModernPOS, ClientBroker and the offline database including all customizations.

**9. Run Modern POS. If you see an error about UAC being disabled, enable it, reboot and continue**

**10. After you activated ModernPOS, go into AX and enable Offline mode for this register.**

**11. Run the job 1070 against the channel data group you are using (demo: Houston)**

**12. Verify in "Download Sessions" that 1070 succeeded**

**13. Logoff ModernPOS and logon again.**

**14. In ModernPOS, go to "Database Connection status" and wait till the "Offline sync status" reflects the fact that Offline is enabled. This may take a few minutes.**

**15. Once all Jobs show as OK, hit the Disconnect button and run through your test scenario.**

#### 4.11.6 Using the RetailServer Test Client

This simple application is very useful to make 1) RetailServer calls, or 2) "offline mode calls" through the RetailProxy or 3) both. It acts as a client, similarly as the POS clients, but requires no UI changes and therefore allows for more rapid testing and development. If you use this tool to verify customizations in the areas of Channel database, RTS, CRT, or RetailServer **before** you hand them off to the UI team.

Note:

- To see a console with errors/logs, use the "Debug" button.

*Table 10 RetailServer Test Client usage steps*

##### Steps

**1. Open the solution file under folder Retail**

**Sdk\SampleExtensions\RetailServer\Extensions.TestClient. Compile and run it with F5**

**2. Enter the RetailServer url in the text box next to the "Activate New" button and click the button (get the url from the IIS management console)**

**3. Enter device and register Ids and hit "Activate" button**

**4. Enter the AAD credentials that has the registration privileges and hit Ok**

**5. Wait a few seconds**

**5. The test client should now show what device is registered.**

**Note: Next time you launch this application, you can just select the previously registered device.**

**7. Hit login button and login with worker credentials.**

**8. Hit Default button. This makes a few simple calls to RetailServer**

**9. Add your own (test) code in the "Custom" button handler**

**10. Uncheck the Online Mode check box to call RetailProxy for offline mode (you must have implemented offline mode correctly and ran ModernPOS offline mode before, so the database for offline exists)**

**11. Hit the "Test Offline Adapters" button to verify that all code has appropriate offline mode implementations. Making sure this is the case, will avoid runtime errors when ModernPOS is in offline mode.**

For any further details of this tool, see its source code.

## 5 Application lifecycle management (ALM)

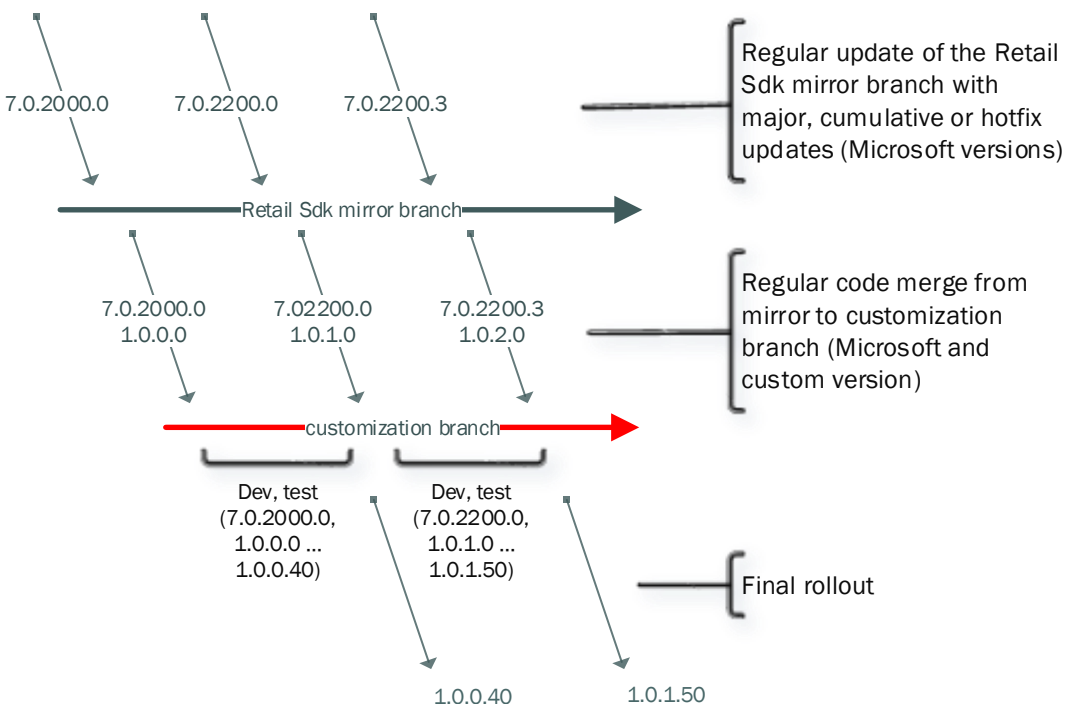
A good ALM solution provides version control, build, automated builds, planning tools, tracking tools, dashboards, customization and more (<https://msdn.microsoft.com/en-us/library/fda2bad5.aspx>). The Retail Sdk is organized in a way that it aligns well with these. Visual Studio Online is a great tool and is recommended. Some of the steps below use VSO with TFS as the basis.

### 5.1 Branching and versioning

In order to work efficiently in a team or even just to be able to go back and look at some changes that were done in the past, it is vital to have a good branching strategy and versioning discipline.

Figure 10 shows a simple branching strategy that may work well for most teams. The version numbers are fictitious.

Figure 10 A branching strategy



### 5.2 Retail Sdk mirror branch

A very important point to emphasize is that the un-customized Retail Sdk should be stored in your source control. It is not necessary to store every version, but the versions that your team would like to snap to, may these be cumulative updates or hotfixes should be added. A simple merge of all changes (additions, changes, deletions) should be done only. No other development work should occur in this branch.

The Retail Sdk has its own version. All included Retail binaries and packages have the same version. That version can also be found in the root of the Retail Sdk in a file named "Microsoft-version.txt".

### 5.3 Customization branch

Once the development starts, a new branch should be created (customization branch). At the beginning of the initial branch-out, it will be an exact copy of the Retail Sdk mirror branch. This is the branch for a team's development.

The version of the customization branch must be incremented at least every time a build is created for testing, or even daily. The file version to increment is defined in Customization.setting with the "CustomVersion" property. If you update it and rebuild, all binaries, packages, manifest files will be updated accordingly. Note that updating the "CustomAssemblyVersion" should only be done when the update is not backwards compatible and/or for major new releases. That should occur very rarely. For example, Microsoft's assembly version stayed the same for AX7's multiple CTP releases.

As there are Microsoft assets and your own changes in the same branch, essentially it has two file versions. The first one is Microsoft's version of the Retail Sdk the current branch is based on, and the second is the "CustomVersion". In the sample in Figure 10, the current file version of the customization branch is 1.0.2.\* (based on Microsoft's 7.0.2200.3). The file version of the first rolled out release was 1.0.0.40 (based on 7.0.2000.0)

When a testing phase is finished and the final packages are being deployed with that version it is important to increment the version (or alternatively create a source control label).

### 5.4 Branching walkthrough for Visual Studio Online

Below is a step-by-step guide for how to onboard the Retail Sdk into a Visual Studio Online (VSO) project. Likely that would be the same project you use for the AX code migration, but it is not a requirement. The assumption is that you already have a VSO project. The following steps would be carried out in order to setup the two branches as discussed above:

#### 5.4.1 Sign into Visual Studio Online and verify server and project connectivity

Launch Visual Studio 2015 and click "Sign in". Use your Microsoft Account that has access to the project. If you have a domain based credential to sign into a domain based TFS, similar steps apply.

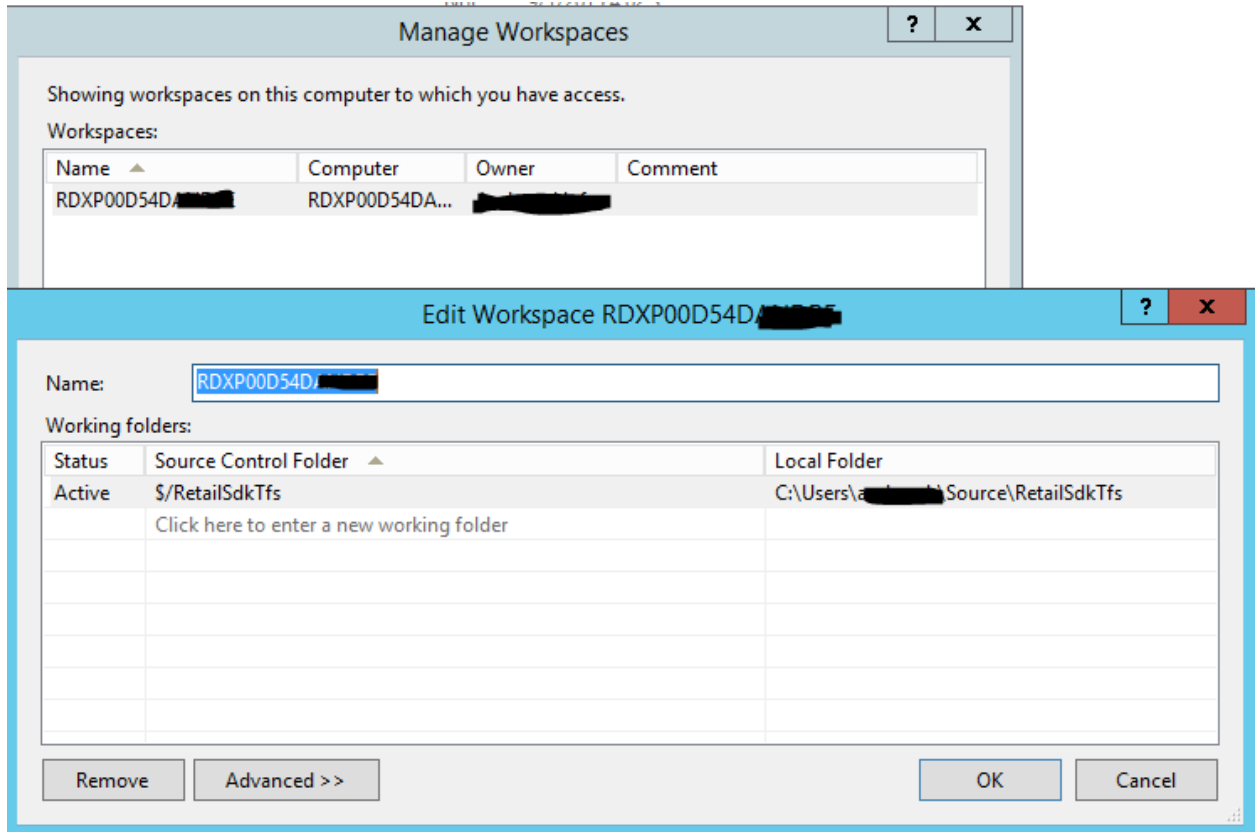
In Team Explorer's Home, click "Manage Connections" and "Connect to a Team Foundation Server". Select the server in the drop down and the project and click Connect.

In Team Explorer's Home, select "Source Control Explorer". You should see the project's structure with any pre-existing branches.

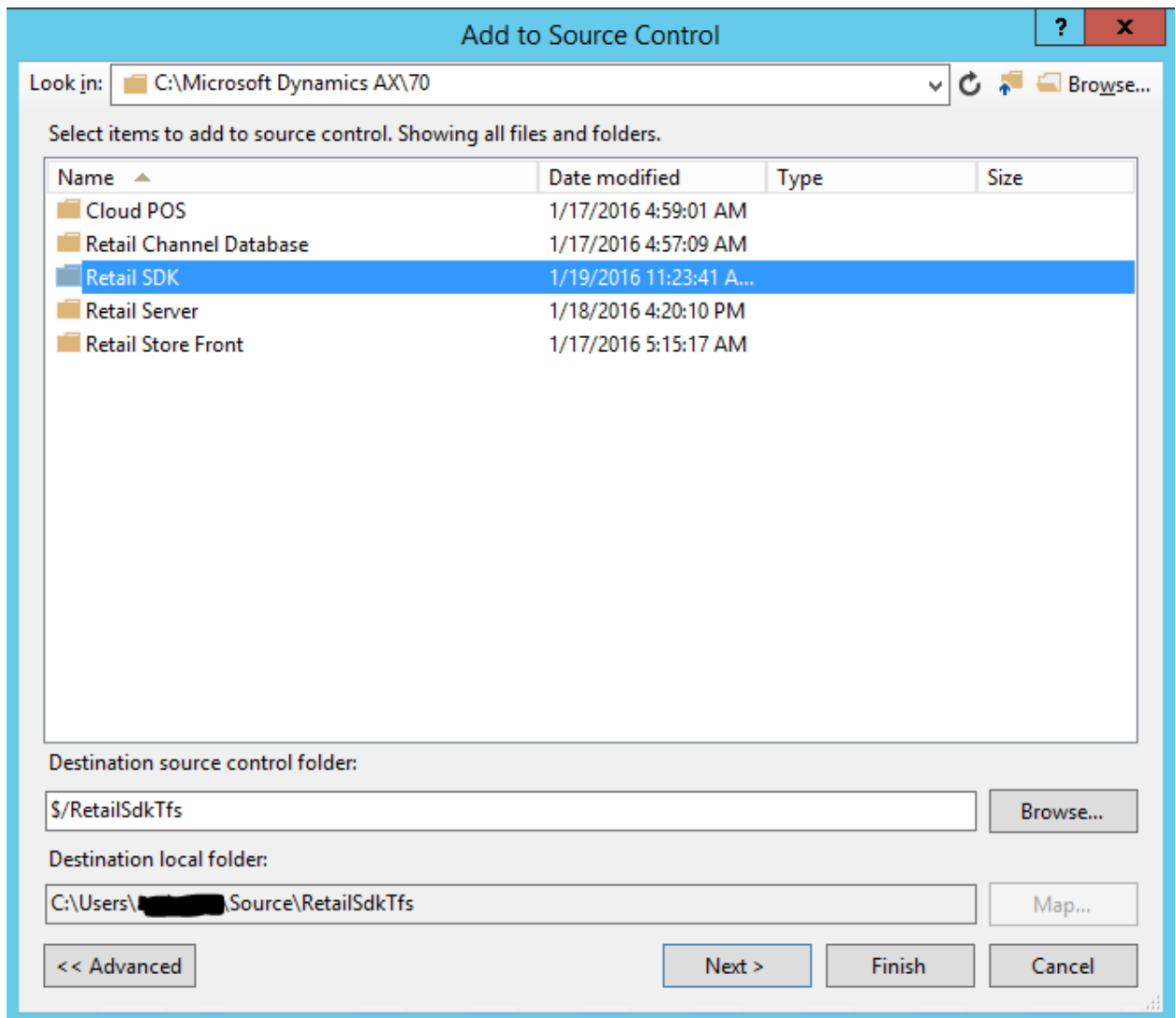
#### 5.4.2 Create the Retail Sdk mirror branch and add the unchanged files

This is a one-time step. It is needed to have a baseline for code merges. Since multiple users could potentially reuse the same developer topology machine, it is strongly advised to use user-specific folders for branches, so multiple users would not work on the same files (this is the default if the following steps are followed).

- If the root of the VSO project is not yet mapped to a local path, do so now. In "Pending Changes", "Actions", "Manage Workspaces", select the one you would like to use and hit Edit. Pick a good source code location in the user folder, i.e. C:\users\\Source\

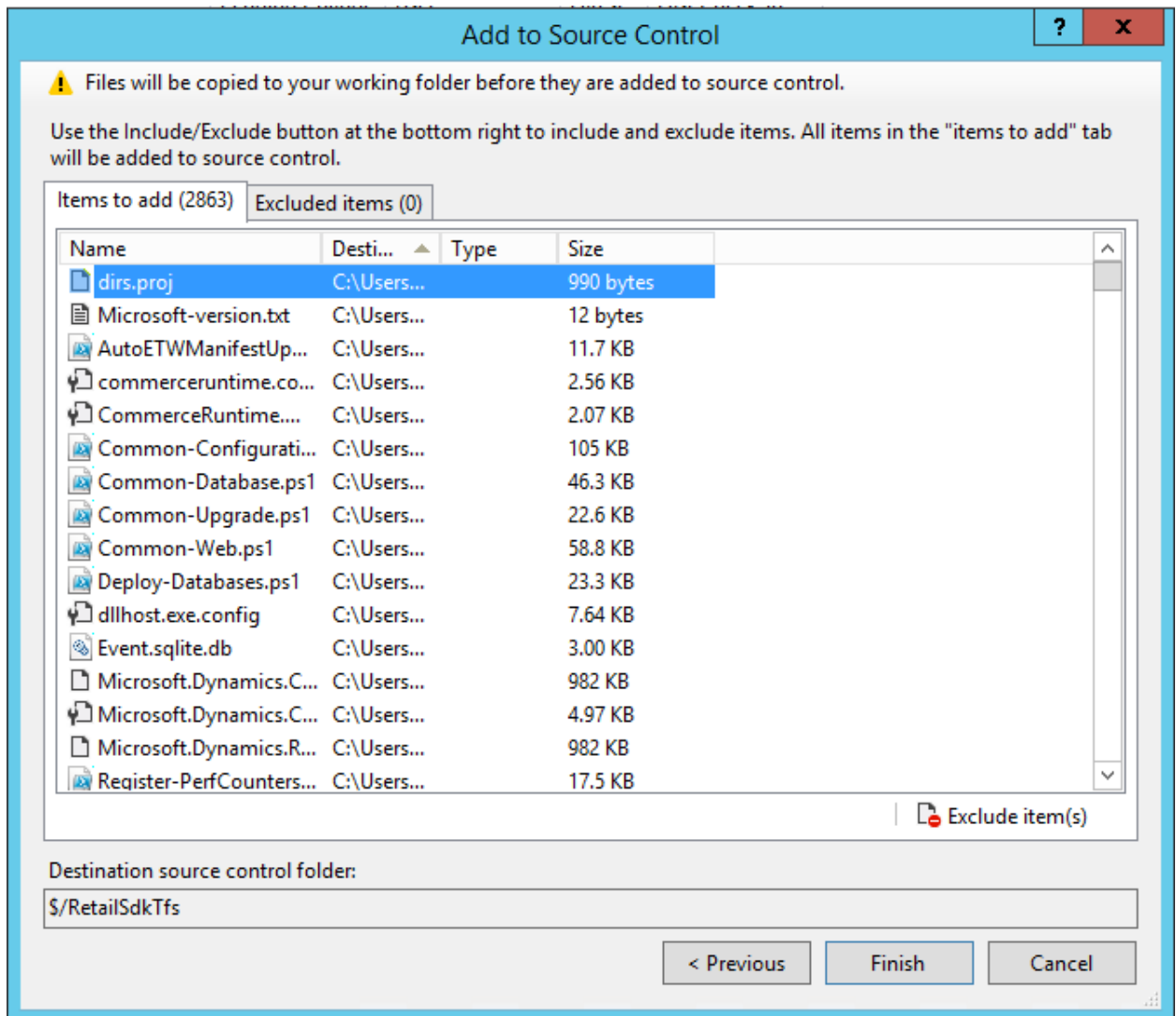


- Add the unchanged Retail Sdk files to source control by right clicking the project in the source control explorer, and selecting “Add Items to Folder”. Pick the original RetailSdk folder, the project, folder to be mapped (note that the new folder will be added to the paths). Hit Next.



- After some calculations, a new window pops up. Make sure that all excluded files are also included by selecting them all, and hitting "Included items". When all files are included, hit Finish.





- The new folder should now be added in the "Source Control Explorer". Make sure you have only the wanted changes in the "Included Changes", no files left in the "Excluded Changes". Then Checkin the changes.
- It will take a few minutes to submit the files.
- In "Source Control Explorer", right click the new folder and rename it to make it clear that this is the mirror branch. For example, call it "Retail Sdk - mirror". Checkin this change.
- Convert this folder to a branch. Right click in the folder, "Branching and Merging", "Convert to Branch...", add a description and hit Ok.

#### 5.4.3 Create the customization branch

This is a one-time step. All that is needed is make a copy of the mirror branch, and create a new branch for it.

- In "Source Control Explorer", right click the mirror branch, select "Branching and Merging...", "Branch". Give the customization branch a meaningful name and description and hit Branch.

Branch from Retail SDK - mirror

Source Branch Name:  
\$/RetailSdkTfs/Retail SDK - mirror

Branch from Version:  
By: Latest Version

Target Branch Name:  
\$/RetailSdkTfs/Retail SDK - dev

Description:  
Branched from \$/RetailSdkTfs/Retail SDK - mirror

**i** The new branch will be created and committed as a single operation on the server. Pending changes will not be created. This operation is also not cancelable once it is sent to the server.

Branch Cancel

- The new branch should now be listed in the Source Control Explorer.
- In order to start working on it, right click the new branch and select "Get latest version".

Note: This is the main branch that developers should work in and builds should be created from. Ideally, the mirror branch could be hidden via removing Read privileges in VSO, so nobody changes it by mistake. Only very few people need access to it, mainly to update it before merging into the dev branch.

#### 5.4.4 Updating the Retail Sdk mirror branch

Assume your team is ready to update AX, Retail components and the Retail Sdk. This may be a major update or a hotfix. In order to get the changes into your current working dev branch, you must take the detour via the mirror branch. It will make merging much simpler.

- Close all but one Visual Studio instances
- Delete all files in the mirror branch in Windows explorer, and add the new Retail Sdk back. This will ensure that removed files are properly being removed from the source control.
- In "Source Control Explorer", right click the mirror branch, "Add items to Folder...", Add all folders from the same source location back. Make sure there are no "excluded items", and hit Finish.
- Make sure there are no files from the mirror branch listed under "Team Explorer", "Pending Changes", "Excluded Changes" and "Detected". If there are, promote them to the "Included Changes"
- Check In the changes.

#### 5.4.5 Updating/merging the customization branch with an updated Retail Sdk mirror

- **Make sure you do not have any changed files in the customization branch before you start. If this is difficult to accomplish create a new client mapping, get the customization branch into a different folder or machine and do the merge there. Do not start merging if you have opened files.**
- In Source Control Explorer, right click the mirror branch and select “Branching and Merge...”, Merge
- Make sure that the source is the mirror branch and destination is your customization branch
- Hit Next and Finish
- Resolve any possible merge conflicts
- Watch closely that all “Included files” are the correct files. These should only be the merged files, or updated files in the mirror
- Watch closely that all “Excluded files” only include generated files. Do not promote them
- Check in the changes.

## 6 Getting started writing your customizations

### 6.1 Best practices

#### 6.1.1 Use of source control

The use of source control should be emphasized here, ideally with the user of both a mirror branch and one or multiple customization branches (see 5).

It is really needed for any type of team environment to work efficiently, and additionally, will help any support engineer or solution architect to help troubleshoot issues. The first thing support will ask is “What are your changes?” and a source control system will give you the answer.

It is also helpful for your own productivity. Modern systems allow for shelf sets, so you can store away, and bring back un-committed changes very quickly.

The AX side of your code is likely already in VSO, so ideally you could use VSO for the Retail Sdk as well. However, any other source control system will suffice.

#### 6.1.2 Location of code changes

Theoretically, you can make any code change in-line in the Retail Sdk and accomplish your goal. However, that approach can have big issues down the road. Remember, that at some not so far point in the future your team will snap to a new Retail Sdk build (either update or hotfix). At that point, it is possible that a massive amount of code changes will come down with that update. The result of this is the need of a lot of code merges with possible a many merge conflicts. Merging code is off course a normal and necessary part of ALM, but it can be error-prone. In order to reduce the chance for errors and reduce the cost of these code merges, it is advised to think about beforehand where code changes are being done.

One of the suggested approaches is to create your own CRT and RS extension dlls whenever possible. In fact, all the Retail Sdk samples use this approach. In that case, no code merge needs to be done, and an Sdk update will usually require branch update and re-compilation only.

The nature of the POS solution is slightly different. Here it is sometimes not possible to truly write code outside of the provided code in the Sdk. That is true in particular for the view-related code in HTML, CSS and TypeScript files. The Sdk samples highlight the use of a POS extension project in two of its samples (CrossLoyalty, StoreHours). In these samples, a new project is being added to both the POS solutions and most of the new customization are done in that project.

#### 6.1.3 Use Customization.settings

In section 4.8 it is outlined what should be changed in your environment regarding assembly names, versions, and certificates. If you make use of it correctly, you will avoid name and version clashes between your own builds and other partners customizations. For example, it would be problematic if 2 partners produce the same CRT extension DLL with the same name. Remember, that the CRT and RS both load DLLs via MEF, so it is required to have a different names to uniquely identify them.

A team can come up with their own version scheme, but make sure the customized version is updated before a new build is releases. All DLLs, applications are stamped with the version, so it is simple to identify the version an application (including ModernPOS and CloudPOS) come from, **if the version was set before the build.**

Make sure new extension DLLs are added to the ISV\_\* items in Customization.settings correctly. If you do not, things may work locally on the developer topology machine, but the build packages will be missing files.

## 6.2 Adding new business logic as a CRT extension DLL

The steps required have been vastly simplified from previous versions. It should take less than 5 mins if you are already familiar with Visual Studio.

Follow the steps outlined in 4.10.1 in order to add a new project. Once that compiles correctly and the file gets dropped into the References folder, you are done.

## 6.3 Adding a new RetailServer extension DLL

Follow the steps outlined in 4.10.1 in order to add a new project and update some settings. Make sure it compiles correctly and the file gets dropped into the References folder. Since RetailServer is the main ODATA endpoint, you need to tell the Retail Sdk that this is the new DLL for proxy generation. Follow section 4.6, especially the part that discusses “RetailServerLibraryPathForProxyGeneration”. Once you rebuild any project that includes the proxy generation (Typescript proxy, RetailProxy, POS.Core projects) the new DLL will be used.

Note that some of the changes require you to make changes to the proxy project. The reason is that the RetailProxy generates adapter interfaces for offline mode. These interfaces must have an implementation also. Therefore, there are 2 use cases that cause you implement code in RetailProxy to correctly implement for offline mode:

### **You added a new API/action to an existing RetailServer controller**

In this case, the interface implementation is missing a new function and the RetailProxy project will fail at compile time. That is normal and you must add a similar implementation as for the corresponding RetailServer api you added. See CrossLoyalty sample for a sample about that exact case.

### **You added a new RetailServer controller**

A new interface will be generated for you, but no class implementation. Hence, there will be no compilation error. However, at run time when POS would call into this new functionality in offline mode, you would get a run time error. Therefore, in this case you should implement the full interface in the RetailProxy project. See the StoreHours sample for a deeper explanation.

The RetailServer Test Client tool will help you catch the second case, if you use the “Test Offline Adapters” functionality (4.11.6).

## 6.4 Adding new extension project to POS

Follow the Sdk samples CrossLoyalty and/or StoreHours and learn about how these samples use a POS extension project in order to isolate the new POS code somewhat from the SDK code. If this is a good way in your case, re-use the one of the two extension projects and just implement your own action, operations or base types in it.

## 7 Code implementation scenarios

### 7.1 Commerce Runtime extensibility scenarios

#### 7.1.1 Extension property usage on CRT entities

Extension properties are the simplest and arguably the most useful way to extend entities. You would always strive to use this pattern first, unless something prevents you from using it. Using polymorphism/inheritance in order to add simple data members to entities usually causes more issues than it solves (but sometimes it may be needed, depending on the specific case).

A simple way to add new data to an existing entity is to use extension properties. Under the hood, these are name-value pairs on the entity and not persisted into the database by default. If you need persistence, please see 7.1.2.

In order to add an extension property, this syntax must be used:

```
entity.SetProperty("EXTENSION_PROPERTY_ADDED", true);
```

In order to read the property at a later time, this syntax could be used:

```
bool? property = (bool?)entity.GetProperty("EXTENSION_PROPERTY_ADDED");
```

#### 7.1.2 Extension property usage on CRT entities with persistence

Any extension property you add to an entity stays in memory for the lifetime of the object, and additionally, travels across application boundaries. For example, if you add an extension property in ModernPOS, and then call RetailServer/CRT, the key/value pair will be available in that process as well. Even further, if that entity is sent to AX during an RTS call, the key/value pair is also available in the AX process. However, as already mentioned above, it is not persisted by default.

If you need to persist an extension property, data modelling has to be done to make the right design choices about where the data should live. Same table with new column, new table with a join or some other approach is possible. In the case of a new table with a join (a recommended case that fits most requirements well) the customizer has to find out where all “writes” occur and update the SQL code (stored procedure(s)) and find out where all the “reads” occur and update that SQL code as well (SQL view). A good end-to-end example is the EmailPreference sample. In that sample, a single SQL View and a single SQL stored procedure is changed via customization.

It is important to state that new SQL objects need have proper role permission grants. As a very simple example, a new table that needs to be included in CDX sync jobs must be allowed access by the DataSyncUsersRole. For other roles available, inspect the main SQL script in the Retail Sdk\Database folder.

```
IF (SELECT OBJECT_ID('ax.RETAILCUSTPREFERENCE')) IS NULL
BEGIN
    CREATE TABLE [ax].[RETAILCUSTPREFERENCE](
        // removed . . .
    ) ON [PRIMARY]
    // removed . . .
END
GO
```

```
-- grant Read/Insert/Update/Delete permission to DataSyncUserRole so CDX can function
GRANT SELECT ON OBJECT::[ax].[RETAILCUSTPREFERENCE] TO [DataSyncUsersRole]
GO
GRANT INSERT ON OBJECT::[ax].[RETAILCUSTPREFERENCE] TO [DataSyncUsersRole]
GO
GRANT UPDATE ON OBJECT::[ax].[RETAILCUSTPREFERENCE] TO [DataSyncUsersRole]
GO
GRANT DELETE ON OBJECT::[ax].[RETAILCUSTPREFERENCE] TO [DataSyncUsersRole]
GO
```

Last but not least, the SQL update script for your customizations needs to be “registered” in the Retail Sdk’s Customization.settings file. For details see 4.8.

### 7.1.3 Extension property usage on CRT request and response types

Similarly to entities, request and response types can be extended.

```
request.SetProperty("BoolPropertyName", true);
response.SetProperty("BoolPropertyName2", true);
```

```
bool? BoolPropertyName = (bool?)request.GetProperty("BoolPropertyName");
bool? BoolPropertyName2 = (bool?)response.GetProperty("BoolPropertyName2");
```

### 7.1.4 Implementing a new CRT service that handles multiple different new requests

It is a common case to implement a new CRT service. First, new request and response classes need to be created.

The new Request type must implement [DataContract] and [DataMember] attributes in order for serialization to work.

```
using System.Runtime.Serialization;
using Microsoft.Dynamics.Commerce.Runtime.Messages;

[DataContract]
public sealed class GetStoreHoursDataRequest : Request
{
    public GetStoreHoursDataRequest(string storeNumber)
    {
        this.StoreNumber = storeNumber;
    }

    [DataMember]
    public string StoreNumber { get; private set; }
}
```

The new Response type looks similar:

```
[DataContract]
public sealed class GetStoreHoursDataResponse : Response
{
    public GetStoreHoursDataResponse(PagedResult<StoreDayHours> dayHours)
    {
        this.DayHours = dayHours;
    }
}
```

```

        [DataMember]
        public PagedResult<StoreDayHours> DayHours { get; private set; }
    }

```

Second, a new CRT service must be created that uses the request and response types. These steps need to be performed:

*Table 11 Steps to implement a new CRT service*

### Steps

#### 1. Implement the new service:

```
public class StoreHoursDataService : IRequestHandler
```

#### 2. Also, implement two members of the interface:

The SupportedRequestTypes member returns a list all requests this service can handle. The execute method is the one that is being called for you by the CRT if a request for this service is being executed.

```

    public IEnumerable<Type> SupportedRequestTypes
    {
        get
        {
            return new[]
            {
                typeof(GetStoreHoursDataRequest),
            };
        }
    }

```

```
    public Response Execute(Request request);
```

#### 3. Update the commerceRuntime.Config file's composition section (or equivalent) in order to register this service, i.e.

```

<add source="type"
value="Contoso.Commerce.Runtime.StoreHoursSample.StoreHoursDataService,
Contoso.Commerce.Runtime.StoreHoursSample" />

```

It is important to note, that single types or all types from an assembly can be registered. The CommerceRuntime engine will find all IRequestHandler derived types. Here are two registration examples:

```

<add source="type" value="Contoso.Commerce.Runtime.CrossLoyaltySample.
GetCrossLoyaltyCardRequestTrigger, Contoso.Commerce.Runtime.CrossLoyaltySample" />

```

```

<add source="assembly" value="Contoso.Commerce.Runtime.Services" />

```

#### 4. (Optional) Use the CommerceRuntime Test Host to test execute your service

### 7.1.5 Implementing a new CRT service that handles a single new request

It's slightly simpler to create a single-request service.

```

public class CrossLoyaltyCardService :
SingleRequestHandler<GetCrossLoyaltyCardRequest, GetCrossLoyaltyCardResponse>

```

Registration is the same as above.



7.1.6 Implementing a new CRT service that overrides functionality of existing request  
Envision, the request and response types are sufficient but the service implementation must be changed. If some new data needs to be transmitted, you could also extend entity, request or response objects by help of extension properties.

In this scenario, the service can be created the same way as discussed above, with existing `IRequestHandler` types used. Additionally, the registration in the `commerceRuntime.Config` file must precede registration of the service to be overridden. This is important as this is how MEF loads the extension DLLs, the types higher in the file win.

### 7.1.7 Implementing a new CRT entity and use it in new CRT service

Any new entity must be of type `CommerceEntity`. By doing so, a lot of low-level functionality is taken care for you automatically. Below is a sample (taken from the `StoreHours` sample) that illustrates how to create an entity that is bound to the database table. This is the usual case.

```
public class StoreDayHours : CommerceEntity
{
    private const string DayColumn = "DAY";
    private const string OpenTimeColumn = "OPENTIME";
    private const string CloseTimeColumn = "CLOSINGTIME";
    private const string IdColumn = "RECID";

    public StoreDayHours()
        : base("StoreDayHours")
    {
    }

    [DataMember]
    [Column(DayColumn)]
    public int DayOfWeek
    {
        get { return (int)this[DayColumn]; }
        set { this[DayColumn] = value; }
    }

    [DataMember]
    [Column(OpenTimeColumn)]
    public int OpenTime
    {
        get { return (int)this[OpenTimeColumn]; }
        set { this[OpenTimeColumn] = value; }
    }

    [DataMember]
    [Column(CloseTimeColumn)]
    public int CloseTime
    {
        get { return (int)this[CloseTimeColumn]; }
        set { this[CloseTimeColumn] = value; }
    }

    [Key]
    [DataMember]
    [Column(IdColumn)]
    public long Id
}
```

```

    {
        get { return (long)this[IdColumn]; }
        set { this[IdColumn] = value; }
    }
}

```

Using the new entity in a service is pretty straight forward. As already shown above, create a new service as a derived IRequestHandler. Then either use or return the new entity. For example, reading it from the database and returning it as part of the response:

```

private GetStoreHoursDataResponse GetStoreDayHours(GetStoreHoursDataRequest
request)
{
    ThrowIf.Null(request, "request");
    using (DatabaseContext databaseContext = new
DatabaseContext(request.RequestContext))
    {
        var query = new SqlPagedQuery(request.QueryResultSettings)
        {
            DatabaseSchema = "crt",
            Select = new ColumnSet("DAY", "OPENTIME", "CLOSINGTIME",
"RECID"),
            From = "ISVRETAILSTOREHOURSVIEW",
            Where = "STORENUMBER = @storeNumber",
        };

        query.Parameters["@storeNumber"] = request.StoreNumber;
        return new
GetStoreHoursDataResponse(databaseContext.ReadEntity<DataModel.StoreDayHours>(query));
    }
}

```

Please note that in the above example, the CRT runtime engine will automatically make a query to the channel database via the registered data adapter. It will query a type with the name crt.ISVRetailStoreHoursView, generate a where clause and columns as specific in the code. It is the customizers responsibility to provide the SQL objects as part of the customization.

#### 7.1.8 Adding Pre- and Post-triggers for a specific request

In some cases some processing needs to be done before or after a request is handled, there are two hooks that can be used to execute some additional code. These are called pre- and post- triggers.

In order to create new triggers and associate with a request, you must:

1. Create a new trigger class that implements IRequestTrigger

```
public class GetCrossLoyaltyCardRequestTrigger : IRequestTrigger
```

2. in the IRequest.SupportedRequestTypes property, return the list of requests that this trigger should be executed for:

```

public IEnumerable<Type> SupportedRequestTypes
{
    get
    {

```

```

        return new[] { typeof(GetCrossLoyaltyCardRequest) };
    }
}

```

3. Implement the functions that are called before (pre) and after (post) the request:

```

void OnExecuted(Request request, Response response);
void OnExecuting(Request request);

```

4. Register the class in the commerceRuntime.config file:

```

<add source="type"
value="Microsoft.Dynamics.Commerce.Runtime.Sample.CrossLoyalty.GetCrossLoyaltyCardRequestTrigger, Microsoft.Dynamics.Commerce.Runtime.Sample" />

```

## 7.2 RetailServer extensibility scenarios

### 7.2.1 Adding a new ODATA action to an existing controller

This is the simplest case, a new API for a slightly different use case needs to be added. To make this work, we can add the new action by inheritance. For any changes to the APIs to RetailServer, two items steps to be done:

1. The new action or controller needs to be implemented, and
2. The model factory needs to be overridden, on order to add the new corresponding metadata.

In order to extend an existing controller with a POST action, here is an example from the Retail Sdk:

```

public class MyCustomersController : CustomersController
{
    [HttpPost]
    [CommerceAuthorization(AllowedRetailRoles = new string[] {
CommerceRoles.Customer, CommerceRoles.Employee })]
    public decimal GetCrossLoyaltyCardDiscountAction(ODataActionParameters
parameters)
    {
        if (parameters == null)
        {
            throw new ArgumentNullException("parameters");
        }

        var runtime =
CommerceRuntimeManager.CreateRuntime(this.CommercePrincipal);
        string loyaltyCardNumber = (string)parameters["LoyaltyCardNumber"];

        GetCrossLoyaltyCardResponse resp =
runtime.Execute<GetCrossLoyaltyCardResponse>(new
GetCrossLoyaltyCardRequest(loyaltyCardNumber), null);

        string logMessage = "GetCrossLoyaltyCardAction successfully handled with
card number '{0}'. Returned discount '{1}'.";
        RetailLogger.Log.ExtendedInformationalEvent(logMessage,
loyaltyCardNumber, resp.Discount.ToString());
        return resp.Discount;
    }
}

```

Now, we need to override the model factory:

```
[Export(typeof(IEdmModelFactory))]
[ComVisible(false)]
public class CustomizedEdmModelFactory : CommerceModelFactory
{
    protected override void BuildActions()
    {
        base.BuildActions();
        var var1 =
CommerceModelFactory.BindEntitySetAction<Customer>("GetCrossLoyaltyCardDiscountAction");
        var1.Parameter<string>("LoyaltyCardNumber");
        var1.Returns<decimal>();
    }
}
```

As already outlined earlier in the Retail Sdk section, in order for any clients to use this new customization, the build system needs to be adjusted to generate the RetailServer proxy code for the new model factory. That is a configurational step in the build system.

Lastly, the web.config file needs to be adjusted. This needs to be done in the packaging project for RetailServer in the Sdk, and, optionally if local tests will be done also on the local development topology machine for testing.

```
<add source="assembly" value="Contoso.RetailServer.Sample" />
```

### 7.2.2 Adding a new simple controller for entity

Assume, you have a simple entity and need a controller to fetch the data. Please review the StoreHours sample in the RetailSdk. A new RetailServer controller makes sense, and all the low-level work is done in the CRT (new entity, request, response, service). In order to create a new controller, you derive from CommerceController. An example is shown below. The important items are highlighted. The controller name is important, it must match the name of the entity.

```
[ComVisible(false)]
public class StoreHoursController :
CommerceController<SampleDataModel.StoreDayHours, string>
{
    public override string ControllerName
    {
        get { return "StoreHours"; }
    }

    [HttpPost]
    [CommerceAuthorization(AllowedRetailRoles = new string[] {
CommerceRoles.Anonymous, CommerceRoles.Customer, CommerceRoles.Device,
CommerceRoles.Employee })]
    public System.Web.OData.PageResult<SampleDataModel.StoreDayHours>
GetStoreDaysByStore(ODataActionParameters parameters)
    {
        if (parameters == null)
        {
            throw new ArgumentNullException("parameters");
        }
    }
}
```

```

        }

        var runtime =
CommerceRuntimeManager.CreateRuntime(this.CommercePrincipal);

        QueryResultSettings queryResultSettings =
QueryResultSettings.SingleRecord;
        queryResultSettings.Paging = new PagingInfo(10);

        var request = new
GetStoreHoursDataRequest((string)parameters["StoreNumber"]) { QueryResultSettings =
queryResultSettings };
        PagedResult<SampleDataModel.StoreDayHours> hours =
runtime.Execute<GetStoreHoursDataResponse>(request, null).DayHours;
        return this.ProcessPagedResults(hours);
    }
}

```

Note, that for new entities, we also need to override the factory's BuildEntitySets() as below.

```

[Export(typeof(IEdmModelFactory))]
[ComVisible(false)]
public class CustomizedEdmModelFactory : CommerceModelFactory
{
    protected override void BuildActions()
    {
        base.BuildActions();
        var action =
CommerceModelFactory.BindEntitySetAction<SampleDataModel.StoreDayHours>("GetStoreDaysByStore");
        action.Parameter<string>("StoreNumber");

        action.ReturnsCollectionFromEntitySet<SampleDataModel.StoreDayHours>("StoreHours");
    }

    protected override void BuildEntitySets()
    {
        base.BuildEntitySets();

        CommerceModelFactory.BuildEntitySet<SampleDataModel.StoreDayHours>("StoreHours");
    }
}

```

## 7.3 RealTime Transaction service extensibility scenarios

### 7.3.1 Implementing a new Retail transaction service extension method in AX

1. Launch Visual studio
2. In the menu bar click Dynamics 'AX 7' and Select Model management > Create model
3. In the create model window provide the below details:

Model name: Contoso  
 Model publisher: Contoso  
 Layer: USR (Select the relevant layer)  
 Version: 1.0.0.0  
 Model display name: Contoso

**Create model**

**Steps**

- Add parameters**
- Select package
- Select referenced packages
- Summary

**Add parameters**

Model name: Contoso

Model publisher: Contoso

Layer: usr ▼

Version: 1.0.0.0

Model description: Contoso

Model display name: Contoso

Back Next Cancel

4. Click Next
5. In the select package, choose Select existing package and then in the drop down select Application Suite

Create model

Steps

Add parameters

Select package

Select referenced packages

Summary

Select package

☐ Create new package

Create a model that builds into its own assembly and is deployed as a separate package. Choose this option to extend the application.

☒ Select existing package

Create a model that is part of an existing package. Choose this option if your model requires customizations of an existing package, including over-layering of source code and metadata.

ApplicationSuite

Back

Next

Cancel

6. Click Next

**Create model**

**Steps**

- Add parameters
- Select package
- Select referenced packages
- Summary**

**Summary**

Name: Contoso

Display Name: Contoso

Publisher: Contoso

Layer: usr

Version: 1.0.0.0

Description: Contoso

Referenced packages: ApplicationPlatform, ApplicationFoundation

< ||| >

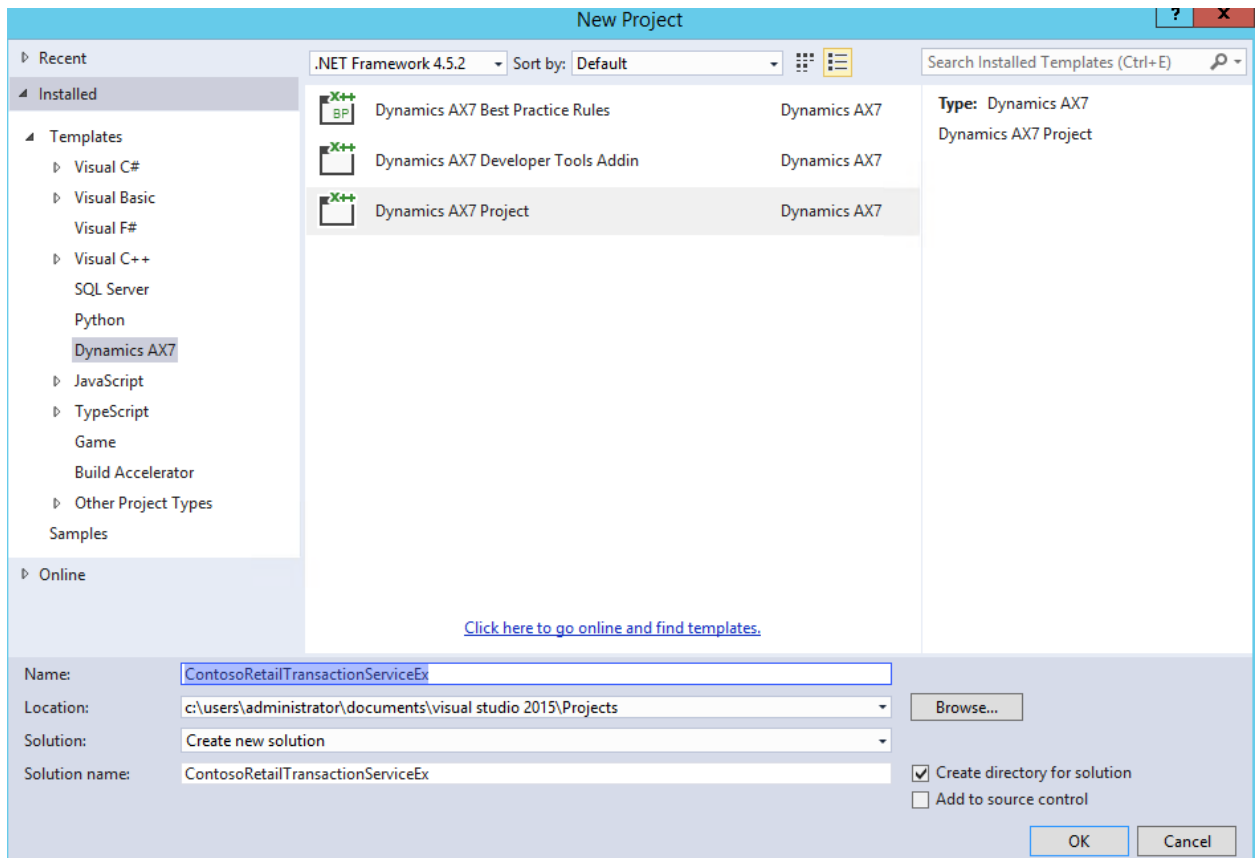
☒ Create new project

☒ Make this my default model for new projects

Back Finish Cancel

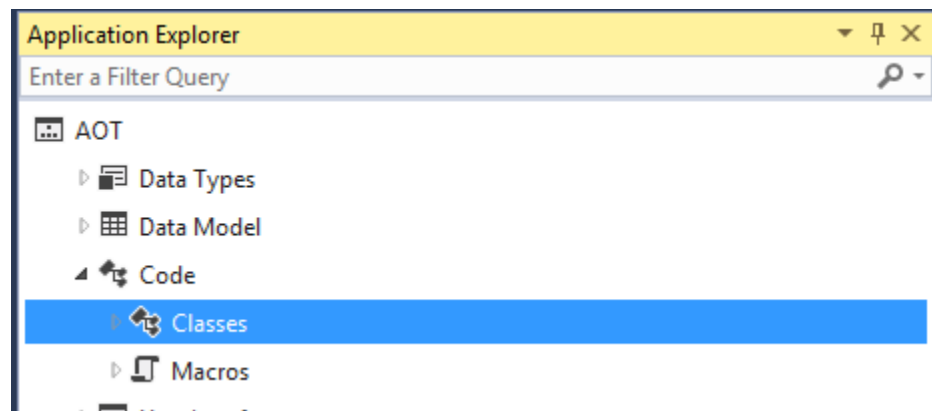
7. Click Finish
8. In the New project windows enter your project name as ContosoRetailTransactionServiceEx



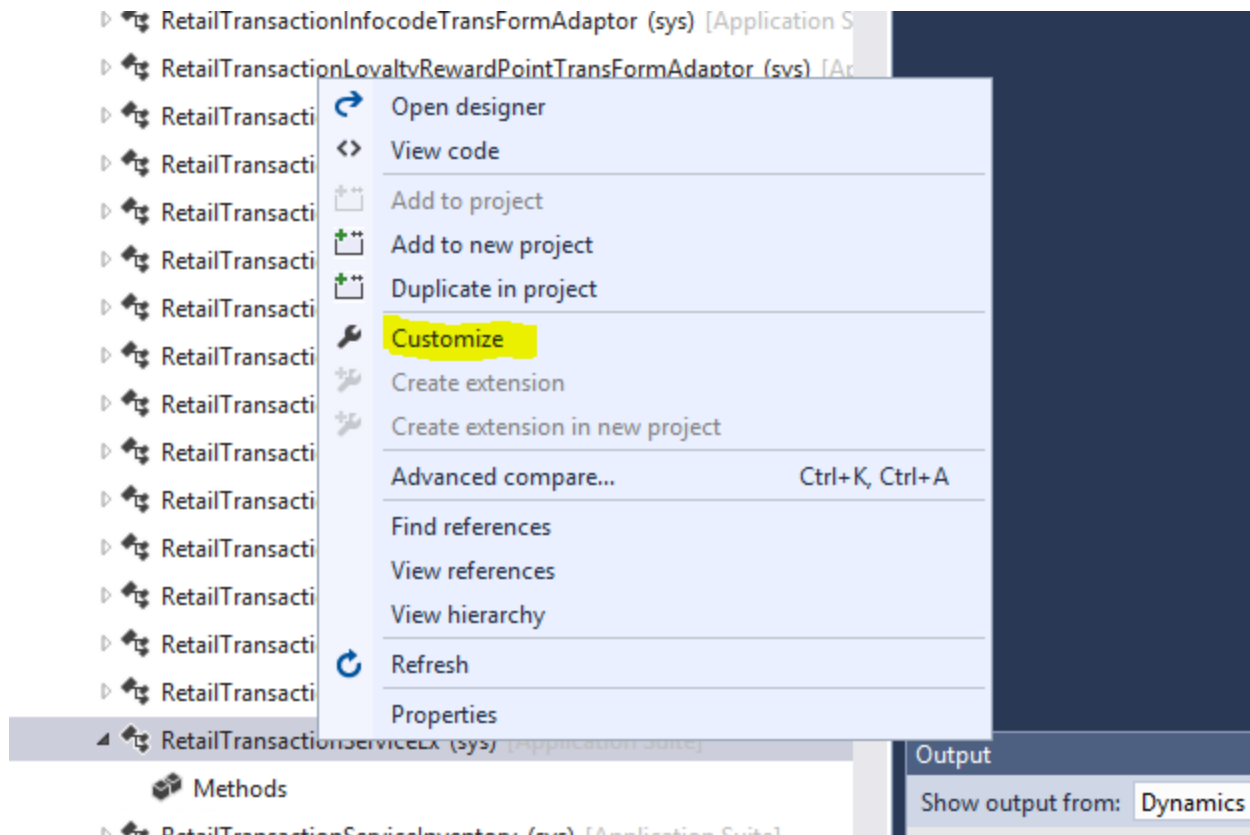


9. Click OK.

10. In the Application Explorer (AOT) Expand the Code node and select Classes



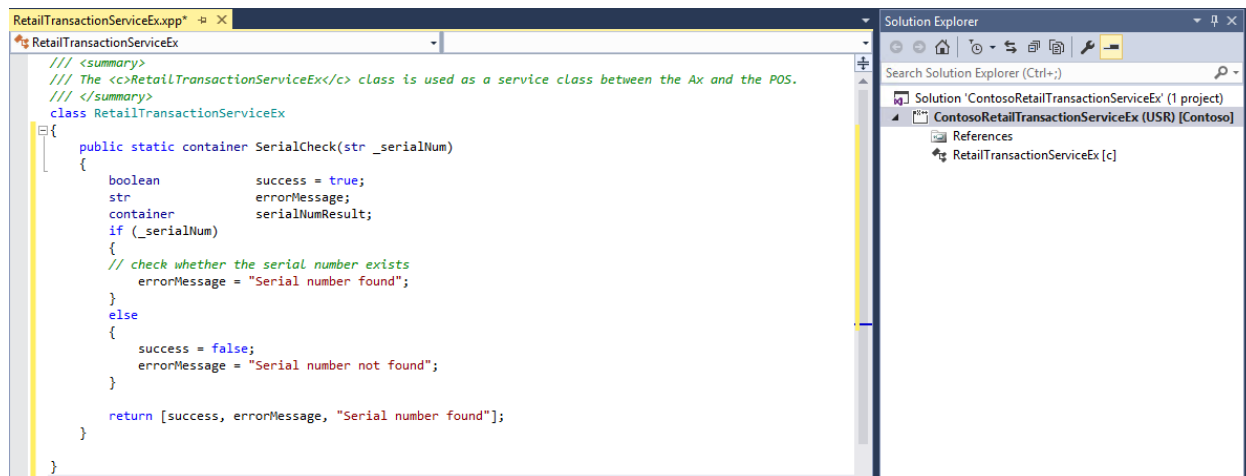
11. In the classes right click the RetailTransactionServiceEx and select Customize



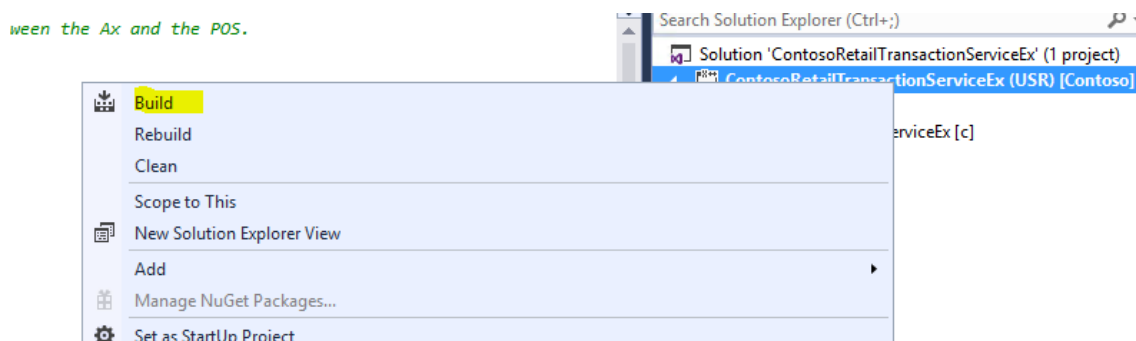
12. In the code editor copy the below sample code:

```
public static container SerialCheck(str _serialNum)
{
    boolean          success = true;
    str              errorMessage;
    container         serialNumResult;
    if (_serialNum)
    {
        // check whether the serial number exists
        errorMessage = "Serial number found";
    }
    else
    {
        success = false;
        errorMessage = "Serial number not found";
    }

    return [success, errorMessage, "Serial number found"];
}
```



13. In the solution explorer right click the project and click Build



After build your new extension methods will be deployed in AX.

### 7.3.2 How to call the new retail transition service method from CRT:

To call the new RTS method, please follow the below steps:

1. Add reference to the `Microsoft.Dynamics.Commerce.Runtime.TransactionService.dll` in your CRT project if not already added
2. Use the below sample code to call the new method:

```

TransactionServiceClient transactionService = new
TransactionServiceClient(request.RequestContext);

ReadOnlyCollection<object> serviceResponse =
transactionService.InvokeExtensionMethod("SerialCheck", "123");

```

From the `serviceResponse` object you can read the response values from RTS.

**Note:** The `InvokeExtensionMethod` method takes 2 parameters one is the RTS method name and other is the list of parameters to be used. The RTS method name passed should be same as the method name you created in `RetailTransactionServiceEx` Class.

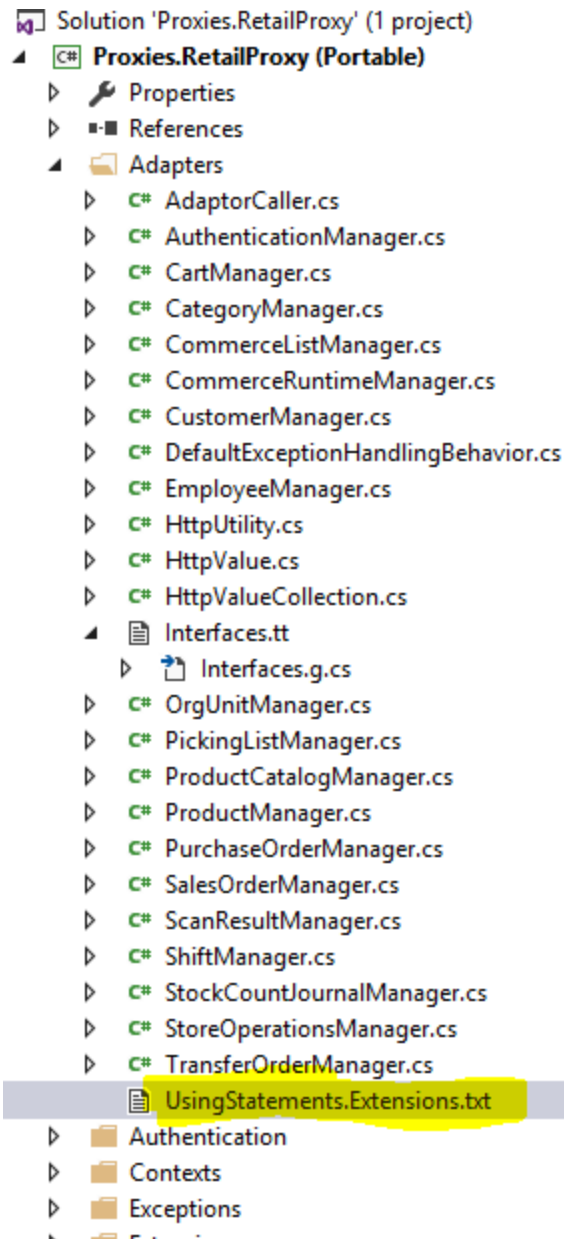
### 7.3.3 Persist Extension properties sent via existing RTS calls

Follow the EmailPreference sample extension, it shows how to do that. No new method is needed, and no new RTS code needs to be written in CRT, as the extension properties automatically flow to AX via the RTS call. Just the part to persist the data is something that still needs to be implemented by the customizer.

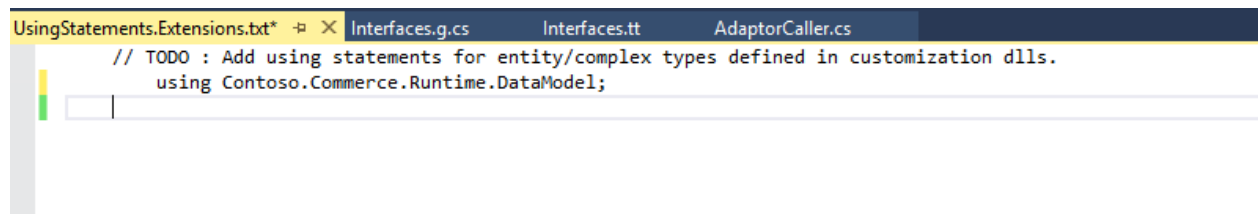
## 7.4 Offline mode extensibility scenarios

### 7.4.1 How to call new/customized CRT service in offline:

1. From Retail SDK\Proxies folder launch the Proxies.RetailProxy.csproj
2. Update the UsingStatements.Extension.txt under the Proxies.RetailProxy\Adapters folder with your new namespace defined for the new entity/complex types introduced in CRT extension projects. These projects or dlls need to be referenced by Proxies.RetailProxy.csproj first.



How to Include your new namespace in the UsingStatements.Extension.txt:



3. After that compile the Proxies.RetailProxy project. After compilation Adapters\Interfaces.g.cs will be updated with your new entity and interface information. Please don't modify anything in this class its auto generated.

4. Add your new manager class in the Adapter folder and call your CRT service from that manager. You can see all the standard entity manager class like SalesOrderManager, PurchaseOrderManager etc are available in the adapter folder.
5. All the libraries used in this project should be portable and signed, if you referencing any custom CRT service it should be portable library. Also update the public key in the commerce runtime config file because after compilation new public key is generated.
6. Build and replace the new proxy dll in the Microsoft Dynamics AX\70\Retail Modern POS\ClientBroker
7. Also drop the custom CRT extension library in the Microsoft Dynamics AX\70\Retail Modern POS\ClientBroker folder (any libraries referenced should dropped in this folder)
8. In the DllHost.exe.config file update the RetailProxyAssemblyName and AdaptorCallerFullTypeName with the new proxy dll and adapter name.

```
<add key="RetailProxyAssemblyName" value="Contoso.Commerce.RetailProxy" />
  <add key="AdaptorCallerFullTypeName"
value="Contoso.Commerce.RetailProxy.Adapters.AdaptorCaller" />
```

9. Restart the dllhost.exe
10. Suggestion is also to follow sections 4.11.5 and 4.11.6 to verify the customization. The advantage of 4.11.6 is that the POS changes do not have to be made yet, but the offline code is still being exercised.

For feedback on errors or suggestions for this document, email [andreash@microsoft.com](mailto:andreash@microsoft.com), [meeram@microsoft.com](mailto:meeram@microsoft.com) or [mumani@microsoft.com](mailto:mumani@microsoft.com).