Microsoft DynamicsTM CRM 2015 Performance Toolkit

# Overview

The Microsoft DynamicsTM CRM 2015 Performance Toolkit was created by the Microsoft CRM product team to formalize performance testing of Microsoft CRM 2015. The performance toolkit can be used by the Microsoft CRM partners and customers to collect data to support their CRM deployment decisions.

The toolkit facilitates load testing the performance of CRM deployments. By carefully planning the required dataset that the deployment needs to support and the workload requirements, the toolkit can be used to test if the scale requirements of a particular deployment can be met. This mechanism can be used in the decisions on a particular deployment solution and avoid costly downtimes at a later stage.

The Performance toolkit contains various tools listed below that can be used in customizing the CRM installation, populating the necessary semantic data for the required scale and conducting the benchmarking tests against the CRM installation. The tools provided in the Performance toolkit are:

1. CRM\_Perf\_Benchmark Tool
2. Data Population Tool
3. ImportCustomization Tool

The Performance Toolkit is distributed in source format under the Open source licensing agreement. Knowledge of Microsoft Visual Studio® and Microsoft Visual C#® is required to use the toolkit. The following are the pre-requisites to conduct the performance tests.

* Microsoft Visual Studio 2013 Ultimate running on a client system used to drive the CRM product. A workstation with 1GB RAM is sufficient to run the benchmarking tests.
* Microsoft CRM 2015 deployment suitable for performance testing. Ideally, the hardware configuration should be similar to the actual production environment.

# CRM 2015 Performance Toolkit Components

The toolkit consists of following tools:

## CRM\_Perf\_Benchmark Tool

This is a tool that can be run in the Visual Studio Test infrastructure that comprises of several scenario driven tests that can be used to test the performance of the Microsoft Dynamics CRM deployment under a load simulation.

## DbPopulator Tool

This is an xml driven tool that uses Microsoft Dyanamics CRM 2015 API calls to load the CRM deployment system with pre-requisite data.

## ImportCustomization Tool

This is a tool that uses the Microsoft Dynamics CRM 2015 API calls to import and publish customizations that can be provided using a customization xml file.

# Setting Up the Test Environment

## Pre Requisites

### Install Microsoft Visual Studio 2013 Ultimate

Install Microsoft Visual Studio 2013 Ultimate according to the Visual Studio installation instructions on a dedicated client computer.

The Microsoft CRM 2015 Performance Toolkit uses Microsoft Visual Studio 2013 Ultimate as the platform for its load tests. This document assumes the user is familiar with using Microsoft Visual Studio 2013 Ultimate.

The toolkit is configured to run on the **sqlexpress** database on the client machine which is installed during the Visual Studio Team System installation process. The toolkit will work with SQL database as well.

### Creation of test users in the Test Domain

You must create a pool of test users on the domain for use in the performance testing of the Microsoft CRM deployment. Each of these users will be named with a prefix and an index, for instance, in our guide we will have test users named **crmusr1** to **crmusrN** where **N** is the number of test users that the CRM deployment needs to support. In the test environment it is assumed all the test users have the same password.

The following command script can be run to populate domain users crmusr1 to crmusr200 with the password “pass”. This should be by logging into the domain as a domain administrator on any of the machines in the system under test:

C:\> for /L %i in (1,1,200) do net user crmusr%i pass /add /domain

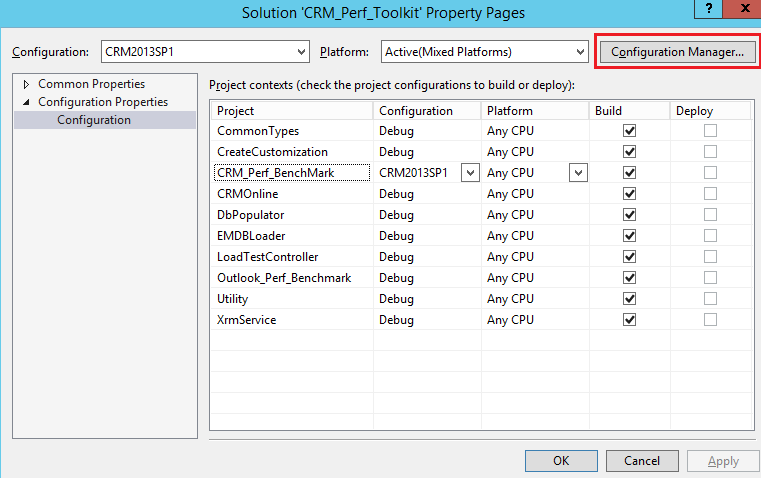
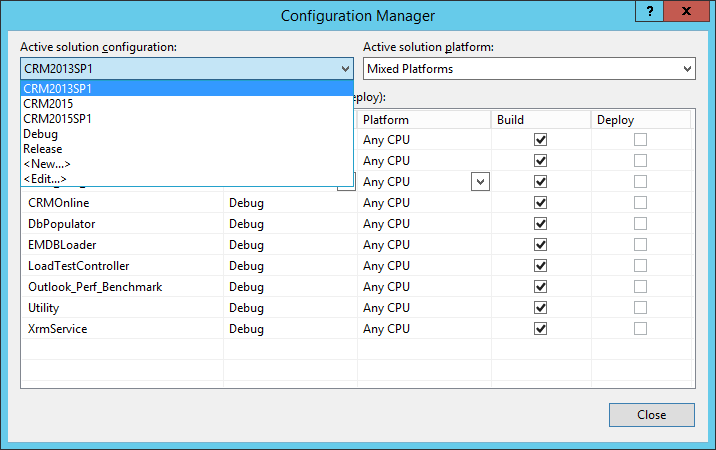
Notes

* The users' passwords must comply with the password complexity policies for your environment. This is one reason we recommend that you use the toolkit in an isolated environment.
* This script uses default settings for things such as password expiration. For more information, see the help for the net user command.

## Microsoft CRM 2015 Performance Toolkit installation

The toolkit zip file is available for download at https://github.com/Microsoft/CRM-Performance-Toolkit

Toolkit Setup:

1. Download toolkit zip file to your local machine
2. Download and install CRM SDK. The version of CRM SDK to install should match the version of the deployed CRM that you will run the load test against. Here is download location for CRM2015 - <https://www.microsoft.com/en-us/download/details.aspx?id=44567>
3. Download and install Windows Identity Foundation SDK V3.5 from <http://www.microsoft.com/en-us/download/details.aspx?id=4451>
4. Download and install .NET Framework 4.5.2 from <http://www.microsoft.com/en-us/download/details.aspx?id=42643>
5. Download and install .NET Framework 4.5.2 Development Pack from <https://support.microsoft.com/en-us/kb/2901951>
6. Unzip the toolkit to the location at your choice
7. Launch Windows Powershell Shell under Administration mode
8. From Powershell shell, run the following commands
   1. cd <toolkit dir>\Setup (<toolkit dir> is the location where you unzip the toolkit to in Step 5)
   2. .\ToolkitSetup.ps1 -adminName:<CRM Administraor Name> -adminPassword:<password> -domainName:<CRM Domain> -crmServerName:<CRM Server Name> -sqlServerName <SQL Server Name> -orgName:<test org name> -configSku:<Online or Onprem> -release <CRM2013SP1 or CRM2015 or CRM2015SP1> -crmSdkBinDir <CRM SDK bin folder path> -crmSDKReservedDllPath <the location where is located> -outlookDllsPath <the location where dlls required to run outlook test cases, needed only if you need to run outlook test cases> -ssl ($true or $false>
   3. Please refer to Readme.txt file in <toolkit dir>\Setup directory for more details regarding toolkit setup.
   4. If the toolkit failed to be compiled at the end of setup, you can compile the toolkit from Visual Studio manually
      1. Open <toolkit dir>\CRM\_Perf\_Toolkit\CRM\_Perf\_Toolkit.sln in Visual Studio
      2. Right click on Solution ‘CRM\_Perf\_Toolkit’ -> Properties
      3. Click on Configuration Manager
      4. 
      5. Select the proper configuration from Active solution configurations dropdown list
      6. 
      7. Click Close -> 0K
      8. Right click on Solution ‘CRM\_Perf\_Toolkit’ -> Build solution

#### Configuration Settings

A configuration setting xml file will be generated at the end of toolkit setup. You can find the configuration setting xml file in <toolkit dir>\Config folder. A sample of the xml that gets created by the setup process is shown below:

<?xml version="1.0"?>

<configsettings>

<authentication type="activedirectory" />

<reloadentitymanager value="1"/>

<ignoretokencheck value="true"/>

<trace value="false"/>

<webservicens value="http://schemas.microsoft.com/crm/2009/WebServices"/>

<crmdomain value="testenvdom"/>

<runas userid="testenvdom\administrator" password="T!T@n1130"/>

<emsqlcnn value="Application Name=CRM\_Perf\_BenchMark;Server=.;Initial Catalog=EntityManager;Integrated Security=sspi"/>

<reportserver value="http://testenvWEB"/>

<parsedependentrequests value="false"/>

<bandwidthcapture value="false">

<netcapdir value="C:\Program Files\Support Tools"/>

<bandwidthoutputdir value="C:\CRMToolkit\CRM\_Perf\_Toolkit\BandwidthTestResults"/>

</bandwidthcapture>

<outlooksyncdir value="c:\outlooksyncdata"/>

<exchangepassword value="password"/>

<outlookuserpercentage value="30"/>

<turboformsenabled value="true"/>

<crmservers>

<crmserver>

<serverbaseurl value="http://testenvWEB"/>

<server value="testenvWEB"/>

<organizationbaseurl value="http://testenvWEB/testOrg"/>

<organization name="testOrg"/>

<organizationserviceurl value="http://testenvWEB/testOrg/XRMServices/2011/Organization.svc" />

<sqlcnn value="Application Name=CRM\_Perf\_BenchMark;Server=testenvSQL;Initial Catalog=testOrg\_MSCRM;Integrated Security=sspi"/>

<configsqlcnn value="Application Name=CRM\_Perf\_BenchMark;Server=testenvSQL;Initial Catalog=MSCRM\_CONFIG;Integrated Security=sspi"/>

<AD>

<userbase value="crmusr"/>

<userpassword value="password"/>

<start value="1"/>

<count value="320"/>

</AD>

</crmserver>

</crmservers>

<executemultiple>

<maxThreads>4</maxThreads>

<batchSize>1000</batchSize>

<totalEntities>10000</totalEntities>

</executemultiple>

</configsettings>

#### Test Users configuration – crmservers/crmserver/AD in ConfigSettings.xml file

#### Test Users BaseName

This is the base name of the test users. The way the toolkit infrastructure is setup is by having a set of test users who have the same base name with an index and who have the same password. As shown in the form above, the base name is the name created for the users in the pre-requisite step.

#### Test Users Password

It is assumed by the toolkit infrastructure that all the test users have the same password which is specified in this section.

#### Test User Start

This is the number that specifies the start index of the test users. It should be specified as a number.

#### Test User Count

This is the number that specifies the number of users from the start index. If start is **201** and count is **200** then the **crmusr201** to **crmusr400** will be used in the toolkit.

All the other configure settings are self-explained.

## Post Installation

### Data Population

After the setup for the CRM Performance Toolkit is completed, the next step before running the performance tests is to create the necessary data. A set of sample xml files that drive the dbpopulator tool are configured for the CRM system and placed in the <Toolkit Install Dir>\Binaries\XMLFiles\DBPop folder. A set of sample customization xml files are also placed in <Toolkit Install Dir>\Binaries\ImportCustomization folder.

If the users want to get started with these pre-configured sample xml files, which create the data that the Microsoft CRM product team has used in their performance testing, they can simply run the “DataPopulation\_320UserBaseline.ps1” powershell script. This will do followings:

* customize the Microsoft Dynamics CRM 2015 deployment with a few sample customizations
* add the users specified in the setup process to the CRM organization
* Create a comprehensive dataset for each of the users specified in the setup.

**Warning:**

* The data population when run with the DataPopulation\_320UserBaseline.ps1 might take a long time, depending on how many users are chosen. The pre-configured xml files create a comprehensive set of data which goes into Gigabytes in the database. It is advised for the users to read the subsequent parts on creating their own xml files that is more appropriate for their needs. An easier alternative is for the users to open these xml files and reduce the counts of entities that are created for each user. This will create a wide variety of entities at a lesser number per user.
* It is also advisable to back up the empty CRM databases so that it is always easy to revert back to the beginning state of empty CRM system if the data populated is not suitable for their needs.

### Compile the CRM\_Perf\_Benchmark tool

Once the data has been populated, open the CRM\_Perf\_Benchmark solution and compile the solution. Once the solution is compiled, the list of tests each of which is a CRM user action scenario can be viewed by opening the Test Manager.

### Populate the EntityManager Database

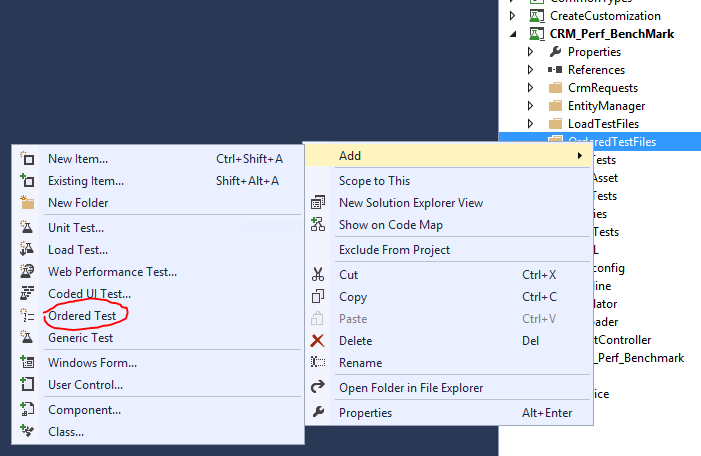
To keep the tests realistic, it is not advisable for the test infrastructure to touch the CRM databases directly during the test runs. For this reason, all the pre-requisite data used by the performance tests is copied to a local database on the Client from the CRM database servers. This database is called EntityManager database. The EntityManager database is created by the CRM Performance Toolkit setup process. The copying of the data from the CRM databases to the EntityManager database can be done running the EMDBLoader.exe which is a part of the CRM\_Perf\_Toolkit solution.

**Note:**

* Once the EntityManager data is copied. It is advised to back up the EntityManager database and the CRM databases. This process ensures that all the databases are in sync. If for some reason, if the user wants to go back to the original state, it can be easily achieved by restoring all the databases back to this state.

### Run a WebTest

With this the user should be ready to run the performance tests. Most configuration and setup problems can be diagnosed by running an individual Web test. To run an individual Web test, do the following:

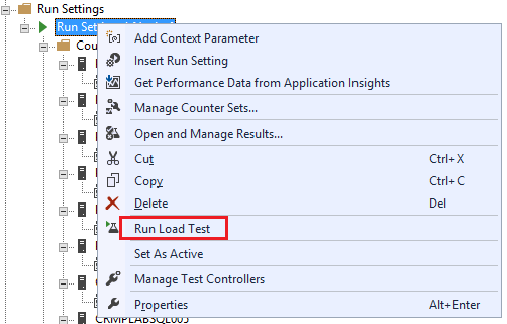
1. From Visual Studio, right click on OrderedTestFiles folder in CRM\_Perf\_Benchmark
2. Click Add -> Ordered Test
3. 
4. Select the web test you want run in Available tests, add it to Selected tests and save the OrderedTest
5. From toolbar, click Test -> Windows -> Test Explorer
6. From Test Explorer, find the ordered test saved in step4. Right click on the OrderedTest -> Run Selected Tests or Debug Selected Tests
7. Use Fiddler or other similar tools to monitor the web requests simulated by the tests

### Running Multiple Tests

You can add more than one tests into an ordered test. An ordered test can be thought of as a container that runs a series of Web tests sequentially.

### Run a Loadtest

1. Open the load test file in Visual Studio
2. Right click on Run Settings -> Run Load Test



# Guidelines for configuring your own performance testing

There are essentially three ways for extending the usefulness of the reference datasets and loadtest provided with the toolkit.

* The first technique is to customize the reference dataset. You can use dbpopulator with custom input files to more accurately reflect a particular customer scenario.
* The second technique is to modify the reference loadtest to test with different user populations, rate of work, and workload composition.
* The third technique is to create new Web tests to cover functionality not included with the reference Web tests.

The results of experimenting with only the reference dataset and loadtest must be interpreted with care because they reflect the performance of a system running a purely hypothetical workload. Although care was taken to make the dataset and workload in the toolkit meaningful, the results do not guarantee any level of performance for a given customer deployment. By taking the time to gather customer workload specifications and customizing the toolkit to more carefully model those specific requirements, confidence can be established early in the sales process and potential design and scalability problems can be identified and managed more effectively.

## Extend the Dataset

Testing the reference dataset and loadtest for larger user populations can be accomplished by the following steps. The descriptions later in this article assume that you have successfully created the reference dataset and executed the loadtestoutlookdm and that the base name for the test users is "crmusr".

1. Back up your Microsoft CRM database, CRM Config DB, and EMDB.
2. Add the users to the domain and to Microsoft CRM. For more information, see the section "Creation of users in test domain" The steps that follow assume that test users crmusr201 through crmusr400 are being tested.
3. Modify the userGroup name line in the NestedSFAEntitiesWithState\_Custom.xml file to populate data for the new users as follows:

…

<userGroup name="{0}\crmusr{1}" startIndex="201" count="200" password="password"/>…

1. The startIndex and count attributes are used to avoid recreating data for users crmusr1 through crmusr200.
2. Use the dbpop tool with the new NestedSFAEntitiesWithState\_Custom.xml input file to create data for users crmusr201 through crmusr400.
3. Back up the new Microsoft CRM and Metabase databases. (This makes it easier to switch between different datasets).
4. Drop and re-create the EntityManager DB. The Entity Manager has to be re-created when you change the Microsoft CRM dataset. See the section "Populate the Entity Manager Database" earlier in this document.
5. Back up the new EntityManager Database.
6. Modify the configsettings.xml file in **ConfigSettings** folder. The new values look as follows:

<crmserver>

…

<AD>

<userbase value="crmusr"/>

<userpassword value="password"/>

<start value="1"/>

<count value="400"/>

<outlookstart value="1"/>

<outlookcount value="200"/>

</AD>

</crmserver>

1. You might need to adjust the test users used in your load test file accordingly. The way to adjust test users in load test file varies depending on the load test mix mode. Please refer to this document for test mix mode reference - <https://msdn.microsoft.com/en-us/library/dd997826.aspx>.

## Create Custom Datasets with dbpop

The dbpop tool works by processing an input XML file that describes what and how many Microsoft CRM entities to create, the data that each of those entities contains, and the relationships between those entities. The simplest way of customizing a dataset is to start with the reference datasets and change the user counts and/or the individual entity counts contained in the reference input XML files. If more significant changes are required, custom input XML files can be created. Several atomic input XML files are provided in the **<Toolkit Install Dir>\CRM\_Perf\_Toolkit\dbPopulator\Sample\_Xmls** directory, which can be used to gain a better understanding of the syntax and format of the input XML files, as well as to serve as building blocks for more complex custom input files. An example of a dbpopulator input file is listed below, taken from **CRM\_Perf\_Toolkit\dbPopulator\Sample\_Xmls\unitTest\_Account.xml**:

<?xml version="1.0" ?>

<DbPopConfig>

<exec threadCount="1" />

<userGroups domain="domainname" webserver="webservername" organization=”orgName”>

<userGroup name="{2}{1}@{0}" startIndex="1" count="1" password="PASSWORD"/> </userGroups>

<objectDist>

<account count="10" save\_1="settag(accountid,account,entityid)" save\_2="settype(accountidtype)">

</account>

</objectDist>

</DbPopConfig>

The key elements and attributes are:

### exec

This is Childless element that controls various execution aspects of dbpop. This element currently supports only the threadCount attribute that sets the number of concurrent threads that dbpop will use to populate the specified data.

### userGroups

Container element used to specify information about the system domain, Web server and organization name. The supported attributes are domain, webserver and organization that should be set to reflect the system domain name and Web server name, respectively.

### userGroup

It is Child element of the userGroups element that is used to specify a group of test users for which to create the test data. Attributes of the userGroup element are name, startidx, count, and password. The name attribute defines the base name for the test users. For example, if you have created a test user pool with users named crmusr1 through crmusr200, the name attribute should be 'crmusr'. For more information, see the previous section titled "Create Test Users". The startidx and count are optional attributes that control how many test users to create data for. If data needs to be populated only for one user without an index such as “administrator” then name will be “administrator” and startidx and count can be omitted.

Some examples of various settings for name, startidx, and count are as follows:

The following setting populates data for crmusr1.

userGroup name="{2}{1}@{0}" startIndex="1" count="1" password="PASSWORD"

The following setting populates data for crmusr1 through crmusr100, inclusive.

userGroup name="{2}{1}@{0}" startIndex ='1' count='100' password=’pass’

The following setting populates data for crmtestuser100 through crmtestuser250, inclusive.

userGroup name="{2}{1}@{0}" startIndex ='100' count='150' password=’pass’

The following setting populates data for administrator

name='administrator' password=’pass’

### objectDist

This is the container element that describes the entities to be populated. A full description of the supported child elements and syntax for the objectDist element is outside the scope of this document and should be explored through code inspection and debugging if you want to customize dbpopulator and dbpopulator input files.

Children of the objectDist element represent the Microsoft CRM entities to be created. Each entity you want to create should have a representative element under the objectDist element. For a complete list of the Microsoft CRM entities currently implemented by the dbpop tool, see the switch statement inside the dbPopulator.CrmEntityTemplate.GetCrmEntityTemplate function in dbpopulator.cs.

Attributes are used to control which fields of the entity are assigned to and their values. Generally, you can add any field that exists in the entity as an attribute to the entity element in the dbpop XML. These attributes can be in addition to or can override the entity fields that dbpop creates by default. To determine which fields dbpop populates by default for a given entity, examine the m\_entityXmlString member in the classes derived from CrmEntityTemplate. Internally, reflection is used to map the field name to the appropriate member of the SDK object, so that your attribute name should exactly match the field name in the SDK object (see CRMService.cs). Multipart names are also supported, for example, the regardingobjectid.value field can be set on an appointment element.

The attributes on the elements that are descendants of the objectdist node, and the entity fields specified in the m\_entityXmlString member variables, control which entities are created, the number of those entities created, and the entity field values themselves. Note that attributes specified in the input XML file will override any default values set in the m\_entityXmlString variables. There are four categories of supported attributes/attribute values, as follows:

Meta values (attribute): The only supported Meta attribute is count, which controls the number of entities created for this element.

<account count="10" … />

Text values (attribute value): If you want a given entity field to have a fixed value, you can assign the attribute a fixed string.

<account creditonhold="false" … />

Replacement functions (attribute value): The dbpop tool implements a variety of replacement functions that are evaluated at runtime to allow for dynamically created data. A common use of a replacement function is to insert random strings into the entities being created to provide a distribution of values over the entity field. Replacement functions have the format:  
replace(replacement function, optional replacement function param, … ).

<account name="replace(randomstring,15,40)" … />

Note that in the case of the account.name field, dbpop supplies this information by default in dbPopulator.CrmAccountTemplate.m\_entityXmlString:

<account>

<name>replace(randomstring,10,50)</name>

…

</account>

In this case the input file overrides the default value, and account names created by this input XML are random strings between 15 and 40 characters long.

For a complete list of currently implemented replacement functions, see the methods in dbPopulator.ReplacementFunctions.

Tag functions (attribute value): In order to create realistic datasets it is frequently necessary to pass information from one created entity to another. For example, to create an Annotation entity that is the child of an Account entity, the ID of the parent Account entity is needed when you create the Annotation. Tags and their associated tag functions are the mechanism by which dbpop allows data to be passed from one entity to another. Tags can be either global or dynamic, and are either set or retrieved.

* Global Tags: Global tags are created by using the globalTags, tag, query, column, and entity elements in the dbpop input file, and occur as descendants of the dbPopConfig root element. The following example is taken from the **CRM\_Perf\_Toolkit\dbPopulator\Sample\_Xmls\unitTest\_Appointment.xml** file:

…

<dbPopConfig>

…

<globalTags>

<tag name="allaccountids">

<query>

<column name="accountid" />

<entity name="account" />

</query>

</tag>

…

</dbPopConfig>

This XML fragment creates a tag named allaccountids that contains a collection of the account.accountid values in the system that are visible to the user running the dbpop tool. Global tags are implemented internally using the standard SDK query mechanisms (such as QueryExpression and ColumnSet). For more information, see the function dbPopulator.TagManager.Retrieve in the dbPopulator.cs file. Global tag processing occurs before any entities are created, in other words, they reflect only data that exists in the system before running dbpop.

* Dynamic Tags: Dynamic tags are created by using save\_ attributes on the elements in the objectDist hierarchy and either the settag or the settype functions. The following example, taken from the **CRM\_Perf\_Toolkit\dbPopulator\Sample\_Xmls\unitTest\_Appointment.xml** file, demonstrates setting dynamic tags:

…

<objectDist>

<account count="1" save\_1="settag(accountid,account,entityid)" save\_2="settype(accountidtype)"

…

</account>

…

This XML fragment creates two dynamic tags: accountid and accountidtype. The settag function has the following format:

save\_N="settag(tagname,entity,field)"

where tagname is the name of the tag that must be used by the retrieve functions, entity is the name of the entity, and field is the name of field that will have its value stored in the tag. Any SDK visible field on entity is allowed, as well as the special fields all and entityid. The all field retrieves all fields for the entity, and the entityid field records the unique identifier of the newly created entity. The entityid field exists as a special case because it is the only value for field that does not require an extra round trip to the server to retrieve (its value is returned by the SDK when the create method is called). This has important performance considerations - setting dynamic tags other than entityid will slow down creation considerably.

The settype function has the following format:

save\_N="setttype(tagname)"

Frequently, when you create entities with the SDK it is necessary to explicitly specify the type for a particular field when there is ambiguity as to what the value in a particular field represents. For example, a customerid field may represent either an account or a contact entity. For an example of how the settag and settype functions are used together with a customerid attribute, see the XML code in the file located at CrmDbPopulate\dbPopulator\TU\_XML\unitTest\_SalesOrder.xml.

* Retrieving Tags: Tags are retrieved by assigning the gettag or getdyntag functions to the desired attributes. For an example of using tags, consider the following fragment taken from **CRM\_Perf\_Toolkit\dbPopulator\Sample\_Xmls\unitTest\_Appointment.xml**:

…

<appointment count="1" requiredattendees="gettag(allsystemuserids,getrandom,systemuserid.Value)" regardingobjectid.Value="getdyntag(accountid)" regardingobjectid.type="getdyntag(accountidtype)" />

…

In order to create a robust appointment entity it is required to specify an attendee and a regarding object. Tags allow us to dynamically supply all this information:

* Requiredattendees='gettag(…)': Here we reference the global tag "allsystemuserids" (see the previous code sample, which created a global tag for accountids for an example of creating a global tag) and select a random userid from that collection.
* Regardingobjectid.Value='getdyntag(accountid)': Here we reference the entityid stored when the parent account entity is created (see the previous code sample). When only a tag name is specified for the getdyntag function it is shorthand for retrieving the fast path entityid tag.
* Regardingobjectid.type='getdyntag(accountidtype)': regardingobjectid is another example of a situation where the type of the entity referred to by regardingobjectid.Value is ambiguous. Here we specify explicitly that the type referred to by regardingobjectid.Value is an account.

In summary, tag retrieval is controlled with the following functions:

gettag/getdyntag(tagname,retrieval\_function,retrieval\_field,filter\_value,filter\_value\_map) – retrieves from the tag tagname. The gettag function references global tags and the getdyntag function references dynamic tags. The getdyntag function has a special shorthand notation of getdyntag(tagname) that retrieves the fast path saved entityid.

* retrieval\_function specifies either getrandom or getfiltered. The getrandom function selects a random entity from the collection in tagname. The getfiltered function is the most complex form of tag retrieval and allows selecting a particular entity from the collection represented by tagname.
* retrieval\_field specifies the precise field to retrieve from the tag collection. Use dot notation for multilevel fields, for example, systemuserid.Value.
* Filter\_value and filter\_value\_map control, if retrieval\_function is getfiltered, the filter value used to specify a particular entity out of the tag collection. Filter\_value specifies the field on the in-flight entity to use as the filter when selecting from the tag collection, and filter\_value\_map handles the scenario where the field name on the in-flight entity differs from the field name on the tag entity. **CRM\_Perf\_Toolkit\dbPopulator\Sample\_Xmls\unitTest\_Product.xml** has an example of using all the filter parameters.

The ordering and nesting of objectDist child elements is significant. Consider the following sample objectDist hierarchy:

…

<objectDist>

<account save\_1="settag(accountidtag,account,entityid)" save\_2="settype(accountidtypetag)" count="2">

<contact parentcustomerid.Value="getdyntag(accountidtag)" parentcustomerid.type="getdyntag(accountidtypetag)" count="3" />

</account>

…

</objectdist>

This input ultimately creates eight entities. The above XML can be interpreted as "create two accounts, and for each of those accounts, create three contacts." Ultimately, two accounts, A1 and A2, and six contacts, C1 through C6, are created in the following order:

A1 is created, accountidtag receives the fast path system id of A1, and accountidtypetag contains the type account.

C1, C2, and C3 are created, and getdyntag(accountidtag) returns the system id for A1. getdyntag(accountidtypetag) returns the account type.

A2 is created, and accountidtag receives A2's system id (so the tag accountidtag now contains the system ids of both A1 and A2 – this is significant when dealing with retrieval functions such as getrandom). Because settype overwrites, instead of appending, its tag, accountidtypetag is overwritten with the account type.

C4, C5, and C6 are created, and getdyntag(accountidtag) returns the ID for A2. In the absence of a retrieval function, the get\*tag functions use the last entity in the tag collection to provide the tag values.

The most common state data that is passed using tags is entityid, which is typically is used to facilitate parent-child relationships. The tag functions are optimized to handle this case; both through the fast path entityid argument to settag, and through the corresponding retrieval using getdyntag(tagname), which always retrieves the ID of the last entity that was created by using settag(tagname,entity,entityid). When combined with nesting, this provides a straightforward syntax for expressing a set of objects and their parent-child relationships. The previous XML fragment can be re-written as:

…

<objectDist>

<account save\_1="settag(accountidtag,account,accountid)" save\_2="settype(accountidtypetag)" count="2" />

<contact parentcustomerid.Value="getdyntag(accountidtag,getrandom,accountid.value)" parentcustomerid.type="getdyntag(accountidtypetag)" count="6" />

…

</objectdist>

This XML also creates eight entities, two accounts (A1 and A2), and six contacts (C1 through C6), but with the following differences compared to the previous example:

* The order is different; in this example the order of creation is A1, A2, C1, C2, C3, C4, C5, C6
* Processing of accountidtag is significantly different. Even though both this example and the one before it record the IDs of the accounts that are being created, this example runs a comparatively expensive round trip to the server to retrieve the account.accountid field. The getdyntag function must also be changed to use a retrieval function, and even then you may end up with an uneven distribution of C1 through C6 across A1 and A2. If you used the same format for the tag processing as the previous example, that is using entityid in the settag function and only the tagname with getdyntag, C1 through C6 would all have A2 as their parent, because getdyntag(tagname) only retrieves the last entity in a given tag collection.

### Dbpop tool source code

<toolkit install dir>\ CRM\_Perf\_Toolkit\dbPopulator

### Run dbpop from command line:

* Cd <toolkit install dir>\ CRM\_Perf\_Toolkit\dbPopulator\bin\debug
* NewDbPopulator.exe <data xml file full path> /discoveryserver:<discoverserver name> /webserver:<webserver name> /orgname:<org name> /domain:<domain name> /organizationserviceurl:<organization service url> /threadcount:<threadcount #> /password:<user password> /userbase:<user base>
* All the parameter values required to run dbpop can be found in the ConfigSettings.xml file - <toolkit install dir>\ConfigFiles\ConfigSettings.xml
* The dbpop log file will be generated in the same folder as the one where data xml file is located by default

**Run dbpop for Dynamics CRM Macro Benchmark testing**

* You can find the scripts used by Dynamics CRM Performance Engineering team to populate test data for benchmark testing here - <toolkit install dir>\CRMToolkit\Binaries\Scripts\ps

## Customize the Load Test

Developing a custom load test starts with gathering information about the number of users, the data that each user will own, the business use cases that each user will run, and the frequency at which those use cases occur.

The section "Creating Custom Datasets with dbpopulator" earlier in this document serves as a starting point for developing a customized dataset. Ultimately, the specifications gathered from the customer must be able to be normalized down to the entities that a given user should own in the system and the relationships between those entities. If there are distinct personas in the customer specification with different data requirements, create a dbpop input XML file for each persona. Partition the test user pool into the different personas (for example, Persona1 = crmusr1 through crmusr100, Persona2 = crmusr101 through crmusr120, and so on), make sure that the userGroup element in each dbpop input file specifies the correct test users for that persona, and run dbpop one time for each input file/persona.

Next, determine the set of business use cases that apply to the customer scenario. The Web tests supplied with the toolkit cover a broad range of business use cases, and can be reused or modified as required.

If you must create a new Web test, start by running the business use case in Microsoft CRM while recording the http dialog that occurs between the client (browser) and the server. Microsoft Visual Studio 2013 Ultimate can be used to monitor the http traffic between the server running Microsoft CRM and the client. See the Visual Studio documentation for more detailed information. If you use do the recording facilities integrated with Microsoft Visual Studio 2013 Ultimate, you should first change the following element in the Web.config file on the CRM system from its default value of "On" to “Off”:

…

<configuration>

…

<appSettings>

<add key="AppMode" value="Off"/>

…

Add your new Web test to Microsoft Visual Studio Solution Explorer by right-clicking CRM\_Perf\_Benchmark, and then select Add Web Test.

One limitation of using the Visual Studio infrastructure to record new Web tests for Microsoft CRM is that web service calls are not handled properly by the current version of the Visual Studio recording tools. Because of this, other HTTP debugging proxies may be necessary. A popular and freely available HTTP debugging tool is Fiddler, available at <http://www.telerik.com/download/fiddler>.

Regardless of what method you use to determine the stream of requests that constitute a business use case scenario, the following topics should be considered:

* Dependent requests
* Think times
* Web service calls
* Selecting a Microsoft CRM user for the test
* Using the CrmRequest class
* Adjusting loadtest parameters

### Dependent Requests

A common concept in any Web application is that of dependent requests. A request is issued to the server for some page, such as an .aspx or .html page, and the response contains nested requests that must be processed in order to render the page. The dependent requests can be for static content, such as images, or they can be for dynamic pages, such as frame targets. The HTTP protocol allows for static content to be cached so that the browser can retrieve the file without additional trips to the server.

All the requests generated by the Web tests in the toolkit derive from a base class named CrmRequest in Util.cs. CrmRequest itself derives from WebTestRequest, and, in the CrmRequest constructor, the value of WebTestRequest.ParseDependentRequests is set to false. Therefore, the Web tests that are shipped as part of the toolkit intentionally prevent Microsoft Visual Studio 2013 Ultimate from automatically issuing the dependent requests, thus simulating an environment where all users have their static content cached.

If you want to follow the example set by the Web tests in the toolkit when you develop your own Web tests, you should issue your server requests by using a CrmRequest object, and issue an explicit request for all the dynamic (.aspx or .asmx) requests recorded during the execution of your business use case.

### Think Times

Think times are fundamental to load testing and represent pauses between actions that occur when a user is interacting with the system. Think times can occur with almost any granularity depending on what is being tested. For example, if you were load testing a keyboard driver, you would be concerned with the think time that occurs between keystrokes.

In the toolkit there are two logical places where think time can occur:

* Between the individual requests that consist of a business use case scenario; that is, between the individual requests in a Web test.
* Between successive executions of a business use case.

The Web tests implemented by the toolkit only contain think time between successive executions of a business use case. This is also used to control the frequency at which business use cases are run.

### Web Service Calls

Microsoft CRM 2015 makes substantial use of Web service calls during routine processing. This is a problem working with Microsoft Visual Studio 2013 Ultimate, which does not natively support load testing with Web service calls. For example, if you use the integrated recording functionality in Microsoft Visual Studio 2013 Ultimate, the resulting request stream will not contain any Web service calls even though Web service calls most likely occurred during the recording.

Therefore, if you decide to use the integrated recording tools, we strongly recommend that you cross check the recorded request stream with the request stream recorded by another application, such as Fiddler (<http://www.telerik.com/download/fiddler>), for the same sequence of actions. If you find that you must insert Web service calls into the request stream generated by the integrated recording tools, you can use some facilities provided by the toolkit.

To insert Web service calls, you will have to convert the recorded Web test to a coded Web test. Do this by right-clicking the root node of your Web test in the Web test designer (for example, WebTest1), and selecting Generate Code. Give the coded Web test a name and then open the new .cs file in Visual Studio.

### Selecting a Microsoft CRM User

The requests generated by the integrated tools do not perform impersonation; in other words, the requests will present themselves to the server using the credentials of the logged on user. It is important that the Web tests impersonate one of the designated test users (that is, crmusrN). To do this, add the following code to the new class, replacing ClassName with the name of the new class:

Public void ClassName\_PreWebTest (object sender, PreWebTestEventArgs e)

{

user = EntityManager.Instance.GetNextUser();

e.WebTest.UserName = user["domainname"];

e.WebTest.Password = user["userpassword"];

}

### Using the CrmRequest Class

The toolkit implements a CrmRequest class and several subclasses, including a subclass for most of the URLs used by the loadtest. For a complete list of the URLs, see the file **CRM\_Perf\_Toolkit\CRM\_Perf\_BenchMark\Utilities\URLs.cs**. These classes make it easier to generate HTTP requests with the proper query string and form parameters.

The classes implemented have the advantage of having member variables for the generally used QueryString parameters that simplifies their usage. Using one of the classes derived from CrmRequest also has the advantage of providing basic response validation. Note that using one of the CrmRequest derived classes will set ParseDependentRequests to false for the request.

The toolkit is included with WSDL.exe generated proxy code for the various Web services it uses in the **CRM\_Perf\_Toolkit\CRM\_Perf\_BenchMark\WSDL** directory. In order to call one of the Web services in your custom coded Web test, first instantiate the appropriate proxy object and create a WebTestRequest variable to hold the generated request, as shown in this example taken from CreateNewEntityTemplate.cs:

Ribbon.RibbonWebService ribbon = new Ribbon.RibbonWebService();

CrmRequest wtr = null

Finally, call the desired Web service method, for example, ReloadCommandBar, inside a try{}catch{} block to "hijack" the serialized SOAP request from the .NET Framework, and assign it to the WebTestRequest variable you created earlier:

try { ribbon.ReloadCommandBar(entityName.ToLower(), "Form", formId, referer, newEntityId, "2"); }

catch (crmException x)

{

wtr = ProcessCRMExceptionFromWebServiceCall(x.wtr);

}

yield return (wtr);

With the new infrastructure in CRM 2015 where the organization name is included in the URL path, the toolkit provides infrastructure to add the organization name to the URL before a request is made as shown above.

Note also the parameter passed to the ReloadCommandBar method. Many of the Web service methods take complex parameters, and several helper classes are implemented by the toolkit to help call the various Web service methods. Review the Web tests shipped with the toolkit for more examples.

### Adjusting Loadtest Parameters

The final step in creating a customized load test is to determine the frequency at which, and the relative weighting of, the Web tests that implement your business use case scenarios. Please refer to test mix models available in Visual Studio 2013 Ultimate here - <https://msdn.microsoft.com/en-us/library/dd997826.aspx>, and choose the appropriate one that suits your testing needs. The sample load test files coming with the toolkit are all created using sequential mix mode.

