Smash Service Programming Guide

Version 1.0 – September 17, 2012

Abstract

The Smash service is a client-server system that enables rapid prototyping and development of solutions for social computing scenarios on Windows desktop and Windows Phone mobile platforms. Smash provides a general sharing mechanism for observable collections, and is part of the Project Hawaii SDK.

Contents

[Introduction 2](#_Toc335638977)

[Prerequisites 2](#_Toc335638978)

[Overview of the Smash Service 2](#_Toc335638979)

[Smash Client API 3](#_Toc335638980)

[Using the Smash Service in an Application 3](#_Toc335638981)

[Smash Concepts 4](#_Toc335638982)

[Walkthrough: SmashSample 6](#_Toc335638983)

[Building the Samples 6](#_Toc335638984)

[Running the Samples 6](#_Toc335638985)

[Desktop Sample Code 8](#_Toc335638986)

[Resources 21](#_Toc335638987)

**Disclaimer:** 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.

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

© 2012 Microsoft Corporation. All rights reserved.

Microsoft, Visual Studio, Silverlight, Windows Azure, and Windows are trademarks of the Microsoft group of companies. All other trademarks are property of their respective owners.

# Introduction

The Project Hawaii Smash Service is a client-server system that provides a general sharing mechanism for observable collections. Smash enables rapid prototyping and development of solutions for social computing scenarios on Windows desktop and Windows Phone mobile platforms. It is provided as part of the Project Hawaii software development kit (SDK).

This document provides an introduction to the managed-code interface for the Smash service and walks you through a sample application that uses the service.

# Prerequisites

Before you can build an application that uses the Smash service, you must:

* Install the Project Hawaii SDK.
* Build the Project Hawaii SDK.
* Obtain Project Hawaii authentication credentials.

For information about installation, build procedures, and credentials, see “Hawaii Installation Guide,” which is installed with the SDK and is available on the web, as listed in “Resources” at the end of this document.

In addition, you should be familiar with the following:

* Windows Communication Foundation (WCF)
* Microsoft Silverlight®
* Windows Phone 7 SDK

# Overview of the Smash Service

The Smash service implements a general sharing mechanism for observable collections. By using Smash, applications can easily perform a variety of social computing tasks:

* Organizing and managing social computing scenarios, such as meetings and chat sessions.
* Sharing files, slides, notes, images, chats, and other information.
* Recording interactions.
* Synchronizing shared data with all participants.
* Storing data in the cloud.

Smash is deployed as a web service that runs on Windows Azure™ and uses Windows Azure table and blob storage. A Smash application can create multiple Smash sessions, and each Smash session is associated with one or more collections. For example, a Smash application for a monthly book club might involve a collection of links to reviews and author information for the current month’s selection and a collection of member comments on the reading.

Smash is intended for short-term storage of shared data. Each Smash application can store a total of up to 5 GB of blob data and 2 GB of table data, regardless of the number of sessions it manages. Each session has a maximum lifetime of 30 days. After 30 days, a session and all its associated data are deleted. The cloud service does not notify the client of the deletion because the client is stateless; attempts to access the data simply return errors.

## Smash Client API

The Smash client application programming interface (API) is defined in the **Microsoft.Hawaii.Smash.Client**, **Microsoft.Hawaii.Smash.Client.Common**, and **Microsoft.Hawaii.Smash.Client.Contracts** namespaces. The client API includes classes and interfaces that applications use to create and manage Smash sessions.

The **Microsoft.Hawaii.Smash.Client** namespace defines the interface that Windows desktop and phone applications use to communicate with the Smash service. Internally, the service formats the body of all POST and PUT requests in JavaScript Object Notation (JSON). To simplify programming, methods in the **Microsoft.Hawaii.Smash.Client** namespace generate calls to the REST interface in the correct JSON format.

The **Microsoft.Hawaii.Smash.Client.Common** namespace defines classes that the Smash service instantiates and applications can use.

The **Microsoft.Hawaii.Smash.Client.Contracts** namespace defines classes that implement the service contract between the Smash service and its clients. Windows applications do not use this namespace; it is provided for the convenience of developers who want to create applications for other platforms.

The following table summarizes the primary classes that Smash applications use. Additional classes define event handlers and completion arguments for the asynchronous Smash service calls.

|  |  |
| --- | --- |
| Class | Description |
| **SessionInfo** | Information about a particular Smash session, including owner, attendees, storage space used, and so forth. Defined in **Microsoft.Hawaii.Smash.Client.Common**. |
| **SessionManager** | Methods that an application uses to create and manage Smash sessions. Defined in **Microsoft.Hawaii.Smash.Client**. |
| **SmashRecordBase(Of T)** | Base class for all Smash record types that are used in a **SmashTable** object. Defined in **Microsoft.Hawaii.Smash.Client**. |
| **SmashSession** | Manages and maintains status information about a Smash session. Defined in **Microsoft.Hawaii.Smash.Client**. |
| **SmashTable(Of T)** | Observable collection that contains shared Smash data. Defined in **Microsoft.Hawaii.Smash.Client**. |

## Using the Smash Service in an Application

To use the Smash service in your own application:

* Add required assemblies to the Visual Studio project.
* Reference the namespaces in your source code.
* Set up your authentication credentials.

### Add Required Assemblies

Applications that use the Smash service depend on one or more of the following libraries, which are built as part of the Project Hawaii SDK:

* Microsoft.Hawaii.ClientBase.dll
* Microsoft.Hawaii.Smash.Client.dll
* Microsoft.Hawaii.Smash.ClientDesktop.dll

To add the libraries to your application

1. Build the Hawaii SDK, as described in “Getting Started with the Project Hawaii SDK.”

2. Reference the appropriate DLLs in your Visual Studio project.

For a phone application:

* Microsoft.Hawaii.ClientBase.dll
* Microsoft.Hawaii.Smash.Client.dll

For a desktop application:

* Microsoft.Hawaii.Smash.ClientDesktop.dll

Only one library is required for desktop applications because the Hawaii Client Base files are directly compiled into the ClientDesktop library.

### Reference the Namespaces

The Smash service defines symbols in several namespaces:

* **Microsoft.Hawaii.Smash.Client** defines the **SmashSession** class and related methods, events, and handlers.
* **Microsoft.Hawaii.Smash.Client.Common** defines classes that manage a session record.

For ease of reference, include one or more of the following in your code:

using Microsoft.Hawaii.Smash.Client;

using Microsoft.Hawaii.Smash.Client.Common;

### Set Up Your Authentication Credentials

Your application authenticates itself with the Smash service by using an Azure Data Market (ADM) client ID and secret. If you do not already have ADM credentials, obtain them as described in “Getting Started with the Project Hawaii SDK.” You can use the same ADM client ID and secret for all applications that use Hawaii services.

To set up the ADM credentials in your code, copy the HawaiiClient.cs file from one of the sample applications, add it to your project, and change the values of the following strings to your ADM client ID and secret, respectively:

* **AdmClientId**
* **AdmClientSecret**

In addition to ADM credentials, each Smash application must have an application secret, which is a unique GUID that is known only to the application. You can create a GUID with any GUID generator, such as the one that is included with Visual Studio. Windows Azure uses the application secret to protect access to the application’s stored data. To set up your application secret, replace the value of the following string in the HawaiiClient.cs file with your application’s unique GUID:

* **ApplicationSecret**

## Smash Concepts

Applications that use the Smash service follow a similar pattern:

* An application creates a session and specifies a list of valid participants and a duration for the session, among other parameters. The creator of each session also chooses a meeting token that other participants supply to join the session.
* After the session has been created, additional clients can join and share data with the creator and other participants.
* Only the creator of a session can enumerate, modify, or delete a session, or change the list of attendees.

### GUIDs for Identification

The Smash service uses GUIDs for identification in three different contexts:

* Meeting token
* Session ID or session token
* Management ID

Every Smash session has a *meeting token* that participants must specify to join the session. The Smash sample application illustrates a simple method that uses a 6-character alphanumeric text token similar to an airline record locator. The session creator shares the 6-character text token with other participants. To derive the GUID that is the meeting token, the sample applications XOR the 6‑character text token with a GUID that is known only to the application. Applications can use that same mechanism or any other mechanism; Smash places no additional constraints on the token GUID. The only requirement is that potential participants must receive the token through a side channel, and not through the Smash session itself. For example, a Smash session for a monthly book club might use the author’s last name, XOR its bits to an application-specific GUID, and use the result as the meeting token.

The *session ID*—sometimes called a *session token*—is returned to each participant upon joining a session. The session ID is maintained internally in the Smash API and cannot be used directly by participants other than the session creator.

The *management ID* is a GUID that is known only to the creator of the session. Unlike the meeting ID, it should not be shared with the other participants. The session owner must supply the management ID, the session ID, and the meeting token to enumerate, modify, or wipe a session. Typically an application should create the management ID once and store it in the application’s isolated storage for later reuse.

### Observable Collections and Data Binding

Smash applications share observable collections through the **SmashTable** class. This class is defined as follows:

public sealed class SmashTable<T> :

[ReadOnlyObservableCollection](http://msdn2.microsoft.com/en-us/library/ms668620)<T>, [ISmashTable](mk:@MSITStore:C:\Users\v-pennyo.REDMOND\MSR\Hawaii\Sources\Services\private\services\hawaiisdk\Sandcastle\Help\Project%20Hawaii%20SDK.chm::/html/a3f1b852-f184-4762-8477-6734f0bace08.htm)

where T : [SmashRecordBase](mk:@MSITStore:C:\Users\v-pennyo.REDMOND\MSR\Hawaii\Sources\Services\private\services\hawaiisdk\Sandcastle\Help\Project%20Hawaii%20SDK.chm::/html/3ddeac01-7829-f888-eeb3-950f495bad6e.htm)<T>

**SmashRecordBase**, in turn, has the following definition:

public class SmashRecordBase<T>

where T : class

Thus, each record in the **SmashTable** is an object of an application-defined class. This design provides great flexibility for Smash applications to share files, slides, images, or virtually anything else, up to the 2 GB table storage limit.

Smash applications can also take advantage of Windows Presentation Foundation (WPF) data binding. By using data binding, an application can connect elements in the user interface (UI) to particular data sources. When the data values change, the UI elements that are bound to the data change automatically. The sample Smash applications use this feature extensively to display output. For more information about data binding, see “Data Binding Overview” on MSDN, as listed in “Resources” at the end of this document.

### Data Storage

Smash applications can store data in Azure as any of the following:

* Serialized **SmashTable** data, up to 2 GB per ADM client ID. A **SmashTable** contains zero or more records derived from **SmashRecordBase<T>**. The size of each record is limited to 64 KB. Attempts to add records larger than 64 KB to a **SmashTable** fail.
* Blobs, up to 5 GB per ADM client ID.

# Walkthrough: SmashSample

SmashSample is installed with the Project Hawaii SDK. The sample implements Windows desktop and Windows phone clients that provide a simple interface with which a user can create, join, and participate in a Smash session. Both samples use the same API, but the sample desktop client uses more Smash features than the sample phone client. All the Smash features are available for both desktop and phone clients.

## Building the Samples

SmashSample has two components:

* SmashSampleAppDesktop, which implements a Smash client that runs on Windows desktop.
* SmashSampleApp, which implements a Smash client that runs on Windows Phone.

To compile and run a sample

1. In Visual Studio, open SmashSampleAppDesktop.sln or SmashSampleApp.sln.

2. Open the HawaiiClient.cs file and set the **AdmClientId** and **AdmClientSecret** strings to your ADM credentials. In addition, obtain a GUID and set it as the **ApplicationSecret**. The sample applications use the **ApplicationSecret** as the base GUID from which to derive the meeting token.

3. Save the HawaiiClient.cs file.

4. Build the solution to generate the desktop client.

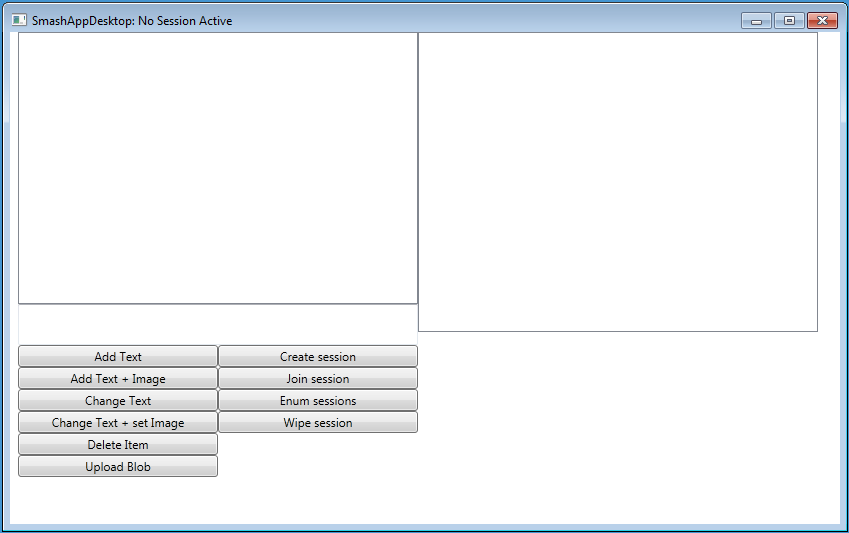
5. Run the sample with or without the debugger, as you prefer.

## Running the Samples

You can concurrently run both the desktop sample and the phone sample, or multiple instances of either sample, to see how the Smash service works.

### Using the Desktop Sample

The following figure shows the initial window for the desktop sample:

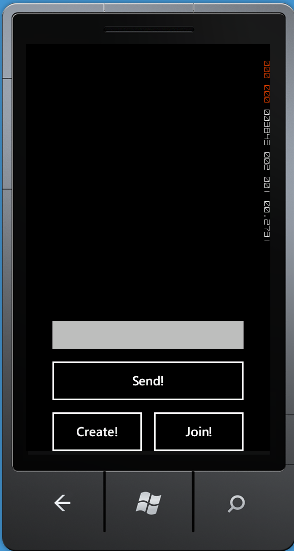


To use SmashSampleAppDesktop

* To create a session, click **Create session**. The sample creates a session, joins the session, and displays the meeting token in the text box immediately above the **Add Text** button.
* To join an existing session that another client created, type the meeting token in the text box and then click **Join Session**. The session’s creator must send you the token by some means other than the session itself, such as email, a text message, verbally, through a Microsoft tag, a barcode, and so on.
* To send a message to the other participants, type the text in the text box and click **Add Text**.
* To add an image to the collection, click **Add Text + Image**. The sample adds a hard-coded image; you can change the image by changing the code.
* To change text or an image, select the item to change in the box on the upper left, then click **Change Text** or **Change Text + set Image**.
* To upload a hard-coded blob to the sample’s storage area, click **Upload Blob**.
* To enumerate all the current sessions that the sample has created, click **Enum sessions**. The session list appears in the box on the right. Each session is identified by a GUID and a session ID.
* To erase a session and all its contents, select the session to erase from the enumerated list, and then click **Wipe session**.

### Using the Phone Sample

The following figure shows the initial screen for the phone sample in the Windows Phone Emulator:



To use SmashSampleApp

* To create a session, tap **Create!**
* To join an existing session, type the meeting token in the text box and then tap **Join!** The session’s creator must send you the token by some means outside the session.
* To send a message to the other participants, type the text in the text box and tap **Send!**

## Desktop Sample Code

The SmashSampleAppDesktop sample demonstrates the following major features of the Smash service:

* Creating a session
* Joining an existing session
* Adding text to a session
* Enumeration of current sessions
* Sharing of images along with text
* Associating an image with text
* Erasing the record of a session
* Uploading a blob
* Changing text

This brief walkthrough introduces the components of the desktop sample. The following table lists the source files for the sample.

### Source Files for SmashSampleAppDesktop

|  |  |
| --- | --- |
| Filename | Description |
| AddItems.cs | Implements methods for the MainWindow class to add text and images to the session. |
| App.xaml.cs | Implements App as the **Application** class object. |
| ChangeItem.cs | Implements methods for the MainWindow class to change text and images in the session. |
| ChatRecord.cs | Implements the ChatRecord class, which manages the data that the session participants share. |
| CreateSession.cs | Implements code for the MainWindow class that creates sessions. |
| DeleteItem.cs | Implements methods for the MainWindow class to delete text or images from the chat box. |
| EnumSessions.cs | Enumerates all the current sessions for the application ID. |
| HawaiiClient.cs | Implements the HawaiiClient class, which stores the ADM and application credentials. |
| JoinSession.cs | Implements code for the MainPage class that creates and joins sessions. |
| MainPage.xaml.cs | Implements the initial UI for the application and manages user input. |
| WipeSession.cs | Erases a session and closes it to futher operations. |
| UploadBlob.cs | Creates and uploads a blob. |

### Create a Session

When you click **Create session**, the sample calls the internal CreateSession\_Click method, which appears in the CreateSession.cs file. The following shows the code for this method:

private void CreateSession\_Click(object sender, RoutedEventArgs e)

{

try

{

string token;

SessionManager sessionManager = new SessionManager();

sessionManager.CreateSessionCompleted += new

CreateSessionCompletedHandler(this.SessionManager\_CreateSessionCompleted);

sessionManager.CreateSessionAsync(HawaiiClient.AdmClientId,  
 HawaiiClient.AdmClientSecret, this.GetMeetingToken(out token),

TextEntry.Text, this.userName, this.userEmail, new string[] { "\*" },

TimeSpan.FromMinutes(60), new Guid(ManagementID), token);

}

catch (Exception ex)

{

MessageBox.Show(ex.ToString());

}

}

CreateSession\_Click proceeds as follows:

1. Declares a local variable for the meeting token.

2. Creates an instance of the **SessionManager** object, which implements the session creation method.

3. Sets SessionManager\_CreateSessionCompleted as the completion handler for the **SessionManager.CreateSessionAsync** method.

4. Creates the session by calling **SessionManager.CreateSessionAsync**. This method is overloaded; the sample passes the following arguments:

* The ADM client ID, which is set in the HawaiiClient.cs file.
* The ADM client secret, which is set in the HawaiiClient.cs file.
* The GUID that is associated with the meeting token. The call to the internal GetMeetingToken method creates both the token and the GUID.
* The name or subject of the meeting. The sample reads this from the first string that the user types.
* The user’s name, which is set to a default value in the MainWindow.xaml.cs file.
* The user’s email address, which is also set to a default value in the MainWindow.xaml.cs file. Each user in a Smash session must have a unique combination of user name, email address, and device name. This mechanism enables the same user to join from different devices—such as a phone and a laptop computer—or on the same device but with different email accounts.

**Note** In your applications, obtain user names, email addresses, and device names from participants. Do not simply use the values that are hard-coded into the samples.

* A string that specifies the names of the users who are allowed to join the meeting. An asterisk (\*) indicates that any user who presents the meeting token can join.
* The duration of the meeting. After the time span has elapsed, the Smash service closes the session and erases all records. There is a small time lag between expiration of the time span and actual erasure of the data, so an application should not rely on exact timing. The maximum duration is 30 days
* A unique management ID that identifies the session creator. Each Smash service client should implement a mechanism to create a random, stableID that is unique for each user. The unique ID ensures the security of the collections and data that the user creates in Smash. In the phone sample application, the internal method GetUniqueManagementID shows how to generate and store such a random ID.
* An application-defined state object. The Smash service passes this object in the arguments to the completion handler that is registered for the **sessionManager.CreateSessionCompleted** event. The sample application passes the meeting token.

When **CreateSessionAsync** is complete, the SessionManager\_CreateSessionCompleted method runs. The following shows the code for this method:

private void SessionManager\_CreateSessionCompleted(object sender,   
 CreateSessionCompletedArgs e)

{

if (e.Error != null)

{

Dispatcher.BeginInvoke(new Action(() =>

{

MessageBox.Show(e.Error.ToString());

}));

}

else if (!e.Cancelled)

{

this.ExecuteJoinSession(e.MeetingToken, (string)e.UserState);

});

}

The Smash service passes two parameters to the completion handler:

* An object (*sender*) that identifies the **SessionManager** instance that called the method.
* A **Smash.Client.CreateSessionCompletedArguments** object that contains information about the completed event. This class inherits from System.ComponentModel.AsyncCompletedEventArgs and has the following GUIDs as additional properties:
* **MeetingToken**, which contains the token for the newly created session.
* **SessionID**, which contains the unique ID for the new session.

The completion handler first checks for errors and displays a message in the UI if an error has occurred. Then, if no errors occurred and the meeting has not been cancelled in the meantime, the completion handler calls the internal ExecuteJoinSession method to join the session.

### Join a Session

When you successfully create a session or click **Join session**, the ExecuteJoinSession method runs. This method appears in the JoinSession.cs file. The following shows the source code:

private void ExecuteJoinSession(Guid meetingToken, string token)

{

if (this.session != null)

{

this.session.Shutdown();

this.session = null;

}

string machineName = System.Environment.MachineName;

if (this.clientInstance > 0)

{

machineName += "\_" + this.clientInstance.ToString();

}

SessionManager sessionManager = new SessionManager();

sessionManager.JoinSessionCompleted += new

JoinSessionCompletedHandler(this.SessionManager\_JoinSessionCompleted);

sessionManager.JoinSessionAsync(HawaiiClient.AdmClientId,

HawaiiClient.AdmClientSecret, this.Dispatcher, meetingToken,

this.userName, this.userEmail, machineName,

new ISmashTable[] { this.chat }, token);

}

ExecuteJoinSession proceeds as follows:

1. Shuts down the current session if one is in already progress.

2. Sets up a unique device ID that contains the client instance and the name of the current machine. The sample increments the client instance each time a new application client starts, which creates a unique device ID so that multiple instances of the sample application can run simultaneously on the device. Smash requires a unique combination of user name, email address, and device name for each participant in a session.

3. Creates an instance of the **SessionManager** object, which implements the methods that a client calls to join a session.

4. Sets SessionManager\_JoinSessionCompleted in the current class to be called when the **SessionManager.JoinSessionAsync** method is complete.

5. Calls **SessionManager.JoinSessionAsync** to join the session. This method is overloaded; the sample passes the following arguments:

* The ADM client ID, which is set in the HawaiiClient.cs file.
* The ADM client secret, which is set in the HawaiiClient.cs file.
* The **Dispatcher** object for the current class. Phone applications must always pass the Dispatcher of the UI thread. For desktop applications, the Dispatcher must be passed only if **SmashTable** objects are used with data binding. If data binding is not used, a desktop application can pass null as a **Dispatcher** and use the **ISmashTable.SyncRoot** object to control locking of **SmashTable** accesses.
* The GUID that is associated with the meeting token.
* The user’s name, which is set to a default value in the MainWindow.xaml.cs file
* The user’s email address, which is also set to a default value in the MainWindow.xaml.cs file.
* The unique client device ID.

**Note** In your applications, obtain user names, email addresses, and device names from participants. Do not simply use the values that are hard-coded into the samples.

* An array with the instance of the **ISmashTable** interface for the sample application’s **SmashTable**<ChatRecord> object instance. ChatRecord is defined as a **SmashRecordBase** type in the ChatRecord.cs file, which is described later in “ChatRecord Class Definition.” Applications can create and associate any number of **SmashTable<T>** objects, provided that the combination of generic type **T** and name of the **SmashTable<T>** objects is unique.
* An object for user state. The sample passes the session token.

When **JoinSessionAsync** is complete, the SessionManager\_JoinSessionCompleted method runs. The following shows the code for this method:

private void SessionManager\_JoinSessionCompleted(object sender,

JoinSessionCompletedArgs e)

{

this.session = e.Session;

Dispatcher.BeginInvoke(new Action(() =>

{

if (e.Error != null)

{

MessageBox.Show(e.Error.ToString());

}

else if (this.session != null   
 && !this.session.SessionID.Equals(Guid.Empty))

{

SmashAppDesktop.Title = string.Format(  
 "SmashSampleAppDesktop: User {0} joined to {1}",   
 userName, this.session.SessionID.ToString());

if (e.UserState != null && e.UserState is string)

{

TextEntry.Text = "TOKEN=" + (e.UserState as string);

}

else

{

TextEntry.Text = this.session.MeetingToken.ToString();

}

}

}));

}

The completion handler stores the SessionID and displays the error string if **JoinSessionAsync** returned an error. If the client successfully joined the session, the completion handler displays an informational message along with the session token.

### Enumerate Sessions

When you click **Enum sessions**, the application calls the EnumSessions\_Click method, which in turn calls EnumSessions\_Core. The following shows the code for this method, which appears in the EnumSessions.cs source file:

private void EnumSessions\_Core()

{

try

{

SessionManager sessionManager = new SessionManager();

sessionManager.EnumSessionsCompleted += new   
 EnumSessionsCompletedHandler(this.SessionManager\_EnumSessionsCompleted);

sessionManager.EnumSessionsAsync(HawaiiClient.AdmClientId,   
 HawaiiClient.AdmClientSecret, new Guid(ManagementID), null);

}

catch (Exception ex)

{

MessageBox.Show(ex.ToString());

}

}

The EnumSessions\_Core method proceeds as follows:

1. Creates a **SessionManager** instance for this request.

2. Establishes SessionManager\_EnumSessionsCompleted as the completion handler for the **SessionManager.EnumSessionsAsync** request.

3. Calls **EnumSessionsAsync** to obtain a list of the current sessions. This method is overloaded; the sample passes the following parameters:

* The client’s ADM ID.
* The client’s ADM secret.
* A GUID that identifies the session owner, which is required to enumerate, change, or erase sessions. This should be a secret, user-specific, unique, stable ID. The phone sample application contains a method that shows how to obtain such an ID. In your own applications, do not reuse the desktop sample code, because others could easily guess the ID, thus creating a security risk.
* An object that contains user state to pass to the enumeration method. No such information is necessary in this example, so the sample passes a null parameter.

When **EnumSessionsAsync** completes, the following completion handler runs:

private void SessionManager\_EnumSessionsCompleted(object sender,

EnumSessionCompletedArgs e)

{

if (e.Error != null)

{

Dispatcher.BeginInvoke(new Action(() =>

{

MessageBox.Show(e.Error.ToString());

}));

}

else

{

Dispatcher.BeginInvoke(new Action(() =>

{

ObservableCollection<SessionInfo> sessionInfos =   
 new ObservableCollection<SessionInfo>(e.SessionInfos);

SessionList.DataContext = sessionInfos;

}));

}

}

The completion handler checks for errors and, if none occurred, displays a list of the current sessions in the SessionList listbox on the right of the UI.

The session list is returned in an **EnumSessionCompletedArgs** object, which inherits from **System.ComponentModel.AsyncCompletedEventArgs**. The **EnumSessionCompletedArgs** object has one additional property—**SessionInfos**, which is also an observable collection. **SessionInfos** is defined as an **IEnumerable** list of **SessionInfo** objects. Each object in the list contains information about a particular session, including the session name and ID, the name of the session owner, the names of the attendees, the duration of the session, and various other details.

In the MainWindow.xaml file, the following line binds the returned data to the SessionList listbox in the UI:

<ListBox Name="SessionList" ItemsSource="{Binding}" Width="400" Height="300" ScrollViewer.VerticalScrollBarVisibility="Auto">

Subsequent lines of XAML format additional details about each enumerated session.

### Add, Change, or Delete Text and Images

The SmashSampleAppDesktop enables you to add, change, and delete the text and images that appear in the chat window.

The AddItemCore method inserts text and an image in the chat record. This code appears in the AddItem.cs source file.

If you click **Add Text + Image**, the sample uses standard Windows methods from the **System.Windows.Media.Imaging** namespace to prepare the bitmap, and then passes the image as a **BitmapSource** object to the internal AddItemCore method. If you click **Add Text**, the sample calls AddItemCore with the text string and passes a null image parameter.

The following shows the source code for AddItemCore:

private void AddItemCore(string text, BitmapSource image)

{

try

{

ISmashTableChangeContext context = this.chat.GetTableChangeContext();

Channels.ChatRecord record = image == null ?   
 new Channels.ChatRecord(DateTime.Now.ToLongTimeString() + " " + text) :   
 new Channels.ChatRecord(DateTime.Now.ToLongTimeString() + " " + text,   
 image);

context.Add(record);

context.SaveChangesCompleted +=   
 new SaveChangesCompletedHandler(this.Context\_SaveChangesCompleted);

context.SaveChangesAsync(null);

}

catch (Exception ex)

{

MessageBox.Show(ex.ToString());

}

}

AddItemCore proceeds as follows:

1. Creates a local variable named context to hold the **ISmashTableChangeContext** interface object that is associated with the **SmashTable** that contains the ChatRecord for the current session.

2. Creates a new ChatRecord that contains a timestamp and the text or text and image.

3. Invokes the **ISmashTableChangeContext.Add** method to add the new ChatRecord to the current record.

4. Sets Context\_SaveChangesCompleted as the **SaveChangesCompletedHandler** for the **ISmashTableChangeContext.SaveChangesAsync** method.

5. Calls **SaveChangesAsync** to save the changes in the ChatRecord.

The sample uses the same completion handler for all add, change, and delete requests. This method simply checks for errors and displays the associated message:

private void Context\_SaveChangesCompleted(object sender,

SaveChangesCompletedArgs e)

{

if (e.Error != null)

{

Dispatcher.BeginInvoke(new Action(() =>

{

MessageBox.Show(e.Error.ToString());

}));

}

}

The code to change or delete text and images is similar, with two exceptions. Where code in AddItem.cs adds a new record to the ChatRecord, the change and delete operations do the following:

* Code in the ChangeItem.cs source file retrieves a changeable copy of the selected record by calling **Channels.ChatRecord.GetUnfrozen**. The **GetUnfrozen** method is inherited from the **SmashRecordBase<T>** class. It then modifies the record with the new information.
* Code in the DeleteItem.cs source file deletes the selected record by calling **Channels.ChatRecord.Delete**. The Delete method is inherited from the **ISmashTableChangeContext** interface.

### Upload a Blob

The Smash service enables users to create and upload blobs of any data type to Windows Azure. When you click **Upload Blob** in SmashSampleAppDesktop, the UploadBlobClick method creates a blob. The following shows the source code from UploadBlob.cs:

void UploadBlob\_Click(object sender, RoutedEventArgs e)

{

SessionManager sessionManager = new SessionManager();

sessionManager.CreateBlobCompleted +=   
 new CreateBlobCompletedHandler(sessionManager\_CreateBlobCompleted);

sessionManager.CreateBlobAsync(this.session, ".jpg", null);

}

The method proceeds as follows:

1. Creates a new session manager object for the request.

2. Establishes sessionManager\_CreateBlobCompleted as the **CreateBlobCompletedHandler** for the **SessionManager.CreateBlobAsync** request.

3. Calls **CreateBlobAsync** and passes the following parameters:

* The current Smash session object.
* The filename extension “.jpg” to store the blob as a JPEG image.
* A null parameter that represents the user state object to be passed to the completion handler.

The **CreateBlobAsync** function creates the blob and sets the **CreateBlobCompletedArgs.BlobUploader** property to a **SmashBlobUploader** object that an application can use to upload data to the blob. The following shows the completion handler for **CreateBlobAsync**:

void sessionManager\_CreateBlobCompleted(object sender,   
 CreateBlobCompletedArgs e)

{

if (e.Error != null)

{

Dispatcher.BeginInvoke(new Action(() =>

{

MessageBox.Show(e.Error.ToString());

}));

}

else

{

StreamResourceInfo resource = Application.GetResourceStream(  
 new Uri("pack://application:,,,/19-baby-tiger.jpg"));

e.BlobUploader.UploadCompleted +=   
 new UploadCompletedHandler(this.BlobUploader\_UploadCompleted);

e.BlobUploader.UploadAsync(resource.Stream, null);

}

}

The completion handler first checks for and reports any errors. If no errors occurred, the method proceeds as follows:

1. Creates a new stream resource for an application-embedded resource that contains the data for the blob to upload. The sample is hardcoded to upload the 19-baby-tiger.jpg file from the sample application resources.

2. Sets BlobUploader\_UploadCompleted as the handler for the **SmashBlobUploader.UploadCompleted** event. An application can also handle the **UploadProgress** event, which provides periodic notification of the number of bytes uploaded and enables the client to cancel the upload operation.

3. Calls the **UploadAsync** method and passes the stream resource and a null parameter, which represents the user state that is passed to the completion handler.

In **UploadCompletedArgs**, the event handler receives a string that contains the URI of the uploaded blob. Applications typically share the URI by using a record in a **SmashTable**. The event handler simply checks for errors and reports this address, as follows:

void BlobUploader\_UploadCompleted(object sender, UploadCompletedArgs e)

{

Dispatcher.BeginInvoke(new Action(() =>

{

if (e.Error != null)

{

MessageBox.Show(e.Error.ToString());

}

else

{

TextEntry.Text = e.BlobAddress;

}

}));

}

### Wipe a Session

When you select a session from the SessionList listbox and click **Wipe session**, the application calls the WipeSession\_Click method. Wiping a session means deleting it and all of its stored data. The following shows the code for this method, which appears in the WipeSession.cs source file:

private void WipeSession\_Click(object sender, RoutedEventArgs e)

{

if (SessionList.SelectedItem != null)

{

try

{

SessionManager sessionManager = new SessionManager();

sessionManager.WipeSessionCompleted += new   
 WipeSessionCompletedHandler(this.SessionManager\_WipeSessionCompleted);

sessionManager.WipeSessionAsync(HawaiiClient.AdmClientId,   
 HawaiiClient.AdmClientSecret,   
 (SessionList.SelectedItem as Session\_Wire).MeetingToken,   
 (SessionList.SelectedItem as Session\_Wire).SessionID,   
 new Guid(ManagementID), null);

}

catch (Exception ex)

{

MessageBox.Show(ex.ToString());

}

}

}

The WipeSession\_Click method proceeds as follows:

1. Checks that you have chosen a session to wipe.

2. Creates a **SessionManager** instance for this request.

2. Establishes SessionManager\_WipeSessionCompleted as the completion handler for the **SessionManager.WipeSessionAsync** request.

3. Calls **WipeSessionAsync** to delete data associated with the session and stop further operations in the session. This method is overloaded; the sample passes the following parameters:

* A string that contains the client’s ADM ID.
* A string that contains the client’s ADM secret
* The GUID that represents the meeting token.
* The GUID that represents the session ID.
* The GUID that represents the owner’s management secret, which is required to wipe a session.
* An object that contains user state to pass to the enumeration method. No such information is necessary in this example, so the sample passes null.

When **WipeSessionAsync** completes, the following completion handler runs:

private void SessionManager\_WipeSessionCompleted(object sender,   
 WipeSessionCompletedArgs e)

{

if (e.Error != null)

{

Dispatcher.BeginInvoke(new Action(() =>

{

MessageBox.Show(e.Error.ToString());

}));

}

else

{

Dispatcher.BeginInvoke(new Action(() =>

{

EnumSessions\_Core();

}));

}

}

The completion handler checks for and displays any errors. If **WipeSessionAsync** completed without error, the completion handler calls EnumSessions\_Core to update the list of sessions.

### [ChatRecord](C:\\Users\\v-pennyo.REDMOND\\MSR\\Templates\\Heading 3) Class Definition

The ChatRecord.cs file implements the ChatRecord class, which manages the data that the session participants share. The definition shows how to:

* Identify which members of the class will be serialized and which will not.
* Ensure that changes are recorded.
* Serialize small images.

The following listing shows the complete definition of ChatRecord, and the notes that follow the listing describe important details:

|  |  |
| --- | --- |
|  | [DataContract] |
|  | public class ChatRecord : SmashRecordBase<ChatRecord> |
|  | { |
|  | [IgnoreDataMember] |
|  | private BitmapSource picture; |
|  | [IgnoreDataMember] |
|  | private string text; |
|  | [IgnoreDataMember] |
|  | private string sender; |
|  | [IgnoreDataMember] |
|  | private string sentTime; |
|  | public ChatRecord(string text) |
|  | { |
|  | this.Text = text; |
|  | } |
|  | public ChatRecord(string text, BitmapSource image) |
|  | { |
|  | this.Text = text; |
|  | this.Picture = image; |
|  | } |
|  | public ChatRecord(string sender, string sentTime, string text) |
|  | { |
|  | this.Text = text; |
|  | this.SentTime = sentTime; |
|  | this.Sender = sender; |
|  | } |
|  | [IgnoreDataMember] |
|  | public string ChatEntry |
|  | { |
|  | get |
|  | { |
|  | return string.Format("{0} ({1}): {2}", this.Sender, this.SentTime,  this.Text); |
|  | } |
|  | } |
|  | [DataMember] |
|  | public string Text |
|  | { |
|  | get |
|  | { |
|  | return this.text; |
|  | } |
|  | set |
|  | { |
|  | this.OnChange(); |
|  | this.text = value; |
|  | } |
|  | } |
|  | [DataMember] |
|  | public string Sender |
|  | { |
|  | get |
|  | { |
|  | return this.sender; |
|  | } |
|  | set |
|  | { |
|  | this.OnChange(); |
|  | this.sender = value; |
|  | } |
|  | } |
|  | [DataMember] |
|  | public string SentTime |
|  | { |
|  | get |
|  | { |
|  | return this.sentTime; |
|  | } |
|  | set |
|  | { |
|  | this.OnChange(); |
|  | this.sentTime = value; |
|  | } |
|  | } |
|  | [DataMember] |
|  | public byte[] ImageBytes |
|  | { |
|  | [System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Security", |
|  | "CA2122:We need to save the image here", |
|  | Justification = "Image needs to be serialized here")] |
|  | get |
|  | { |
|  | byte[] result = null; |
|  | if (this.Picture != null) |
|  | { |
|  | using (MemoryStream ms = new MemoryStream()) |
|  | { |
|  | #if TARGET\_DESKTOP |
|  | JpegBitmapEncoder encoder = new JpegBitmapEncoder(); |
|  | WriteableBitmap bmp = new WriteableBitmap(this.Picture); |
|  | encoder.Frames.Add(BitmapFrame.Create(bmp)); |
|  | encoder.Save(ms); |
|  | #else |
|  | WriteableBitmap bmp = new WriteableBitmap(this.Picture); |
|  | bmp.SaveJpeg(ms, bmp.PixelWidth, bmp.PixelHeight, 0, 85); |
|  | #endif |
|  | result = ms.ToArray(); |
|  | } |
|  | } |
|  | return result; |
|  | } |
|  | set |
|  | { |
|  | if (value != null) |
|  | { |
|  | using (MemoryStream ms = new MemoryStream(value)) |
|  | { |
|  | #if TARGET\_DESKTOP |
|  | JpegBitmapDecoder decoder = new JpegBitmapDecoder(ms, |
|  | BitmapCreateOptions.None, BitmapCacheOption.OnLoad); |
|  | this.Picture = decoder.Frames[0]; |
|  | #else |
|  | WriteableBitmap bmp = |
|  | Microsoft.Phone.PictureDecoder.DecodeJpeg(ms); |
|  | this.Picture = bmp; |
|  | #endif |
|  | } |
|  | } |
|  | else |
|  | { |
|  | this.Picture = null; |
|  | } |
|  | } |
|  | } |
|  | [IgnoreDataMember] |
|  | public BitmapSource Picture |
|  | { |
|  | get |
|  | { |
|  | return this.picture; |
|  | } |
|  | set |
|  | { |
|  | this.OnChange(); |
|  | this.picture = value; |
|  | } |
|  | } |
|  | } |

The following notes apply to the listing:

* At line 1, the **DataContract** attribute enables JSON serialization for the ChatRecord class.
* Line 2 declares the class as a Smash record that is used in a **SmashTable**.
* At line 4, the **IgnoreDataMember** attribute indicates that the picture **BitmapSource** will not be directly serialized. This attribute is required for members and properties that are not directly serialized. Additional examples appear on lines 6, 8, and 10.
* Line 5 declares the internal data member for the image as a **BitmapSource** object. Such an object is not directly serializable to JSON because JSON cannot choose an appropriate format. Instead, the public property accessors at lines 80 and 101 handle serialization.
* Lines 12-26 define the constructors for the ChatRecord class.
* At line 35, the **DataMember** attribute on the Text **string** indicates that the string will be serialized. This attribute is required for every member that will be serialized. All serialized data must be exposed as a property accessor, not directly as a member field.
* At line 44, the **SmashRecordBase<T>.OnChange** method notifies the **ISmashTableChangeContext** interface that the record has been modified. An application must call this base-class method on a **set** accessor. You can see additional examples at lines 57 and 70.
* At line 75, ImageBytes is a public property that exposes the bytes in the image to the JSON serializer.
* At line 80, the **get** accessor encodes the image as JPEG and returns the resulting stream as a **byte**[] array.
* At line 101, the **set** accessor is called with a **byte**[] array and decodes it as a JPEG image.
* Line 125 defines the Picture member, which the sample application and the data binding mechanism use to access the instantiated image. The sample uses small images so that the image data can be serialized inline with the Smash record. The total serialized size must be 64 KB or less, and the use of larger images would exceed this limit. To handle larger images and still take advantage of data binding:
* In the Smash record type, define a member that contains a URI that points to the image.
* Upload the large image by using the **SmashBlobUploader** class.

# Resources

This section provides links to additional information about Project Hawaii and related topics.

Microsoft Research Project Hawaii

<http://research.microsoft.com/en-us/projects/hawaii/default.aspx>

Getting Started with the Project Hawaii SDK

<http://research.microsoft.com/en-US/projects/hawaii/docs.aspx>

Microsoft Research Project Hawaii on Facebook

<http://www.facebook.com/pages/Microsoft-Research-Project-Hawaii/164295863611699>

MSDN

Data Binding Overview  
<http://msdn.microsoft.com/en-us/library/ms752347.aspx>

Programming Windows Phone 7  
<http://blogs.msdn.com/b/microsoft_press/archive/2010/10/28/free-ebook-programming-windows-phone-7-by-charles-petzold.aspx>

How to: Create Your First Silverlight Application for Windows Phone  
<http://msdn.microsoft.com/library/ff402526(v=VS.92).aspx>

Understanding Block Blobs and Page Blobs  
<http://msdn.microsoft.com/en-us/library/windowsazure/ee691964>