How to set up in-app purchases and use the Wallet to store and transfer a virtual currency

Windows Phone 8 introduces new features like the Wallet system and in-app purchases for buying goods for use in your application. This article will explain the practical case of adding a virtual currency to your app, from creating the in-app purchase system to storing the current balance in the Wallet.

Introduction

In-app purchases, or microtransactions, is a billing model that has been gaining traction in the last years as a way for acquiring additional content in videogames. Instead of an up-front payment, the user can download a portion of the game and unlock further levels, items or content by purchasing separate products. The products can be durable, meaning that they are tied to the user's account and remain available even when the game is uninstalled, or consumable, so they get depleted as they are spent during gameplay. One of the most common types of consumable items is a custom in-game currency that allows the player to purchase additional equipment or items for its character.

In this article we are going to learn how to add in-app purchases to a Windows Phone 8 application to obtain different amounts of a virtual currency of our own. To make things easier and more secure, we will store our current balance in the new Wallet system, thus avoiding the hassle of serializing and encrypting the contents to the application's local storage. Finally, we will be able to make dummy purchases using this currency.

Creating the application project in Visual Studio 2012

Open a new Visual Studio 2012 instance and under the **File** menu, select **New > Project**. In this new window, navigate to the Windows Phone templates and create a new **Windows Phone App** project, with a name and location of your liking. When prompted to select the OS target version, choose **Windows Phone OS 8.0**.

Note: If you have trouble locating any of the options specified in the previous instructions, make sure you have version 8.0 of the Windows Phone SDK installed in your machine.

Now, locate the **Solution Explorer** window and open the **Properties** drop-down inside your project's tree. Double click on **WMAppManifest.xml** to display the visual manifest editor and, in the **Capabilities** tab, enable **ID_CAP_WALLET** so our application has the necessary permissions when accessing the Wallet.

In preparation for the next step, build a **Release** version of the app, run it in the emulator and take a screenshot for each of the supported resolutions (by default WVGA, WXGA and 720p, unless you changed them in the **Application UI** tab).

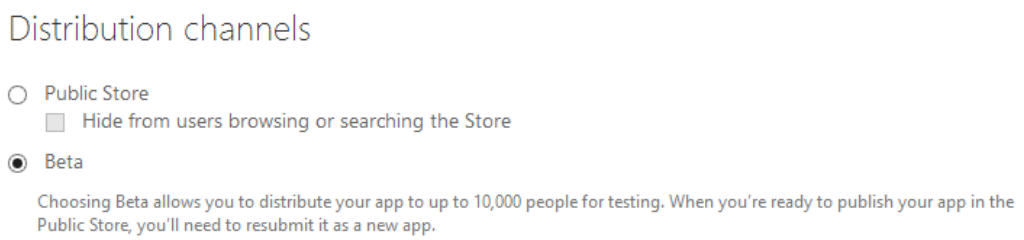
Configuring in-app purchases in Windows Phone Dev Center

Creating a beta release

To enable testing of in-app purchase features, first we must create a beta version of our application and add some purchasable goods to it.

Tip: You can use the Mock In-App Purchase Library instead of creating a beta version of your app and its associated products.

Log in to the Dev Center with your developer account and select the **Submit app** option in the left menu. Fill in the new application's info as you wish and make sure you select the **Beta** option in the **Distribution channels** section (note that you must click in the **More options** drop-down to make it visible).



Tip: If you are going to test the application in a Windows Phone 8 device, make sure you include the device's current Microsoft Account in the appropriate field.

Click on the **Save** button and if everything is correct, you will be redirected back to the **Submit app** page. Now click on the **Upload and describe your XAP(s)** option to continue to the next step.

Click on the **Browse...** link and select the application's previously generated XAP. If everything goes well, the package will be uploaded and you will be redirected to a new page that prompts you to fill in some new properties. If not, an error message will tell you what's wrong with the package (compiled in Debug mode, wrong capabilities specified...). Since you are uploading a beta that won't be available to the general public, fill only the fields marked with an asterisk and upload a random application icon (300x300 px) and the necessary screenshots. Click on **Save** and you will be taken back to the **Submit app** screen. Now, click on **Submit** and your application will be successfully sent.

(C) Copyright Nokia 2013. All rights reserved.

Note: If the **Submit** button is disabled, click on **Show any errors and warnings** to see what's wrong with your application details.

Associating purchasable products

Now we are going to add some products that will be retrieved and purchased by our application. Go back to the dashboard and select **Apps** in the left navigation menu. Navigate the list of existing applications and select the one you want to add the products to. In this view, select the **Products** tab and then **Add in-app product**. You have to fill in the required fields and as many optional ones as you want. Below is a brief description of each field:

- **In-app product alias**: An internal alias for the product to make easier recognizing it.
- **Product identifier**: The ID that your source code will use to query/purchase the item.
- **Type**: Consumable or durable. For virtual currencies we will mark it as consumable.
- **Default language**: If your product doesn't have a description for a specific language, the one defined in the default language will be shown instead.
- **Price**: The cost of purchasing your product.
- **Keywords** (optional): Custom categories/tags that will make finding the item easier.
- **Tag** (optional): additional information about the product.
- **Windows Phone Store languages** (optional): Additional languages in which the product's description will be localized.

Note: You can't select a product price for beta releases: it will always be 0.

Click **Submit** and you will be taken back to the **Add in-app product** page. Select the second step and proceed to fill in the localized properties of your item (product title, description and a 300x300px image). The combo box lets you switch between each available language you specified in the previous step, so you can properly localize your item. When finished, click **Save** and the **Submit**. Again, if the **Show any errors and warnings** text is shown, check for possible mistakes in the data you entered.

Warning: It takes up to 24 hours to get the beta app and products approved for testing. Any calls to in-app purchase functions before approval will likely throw an exception.

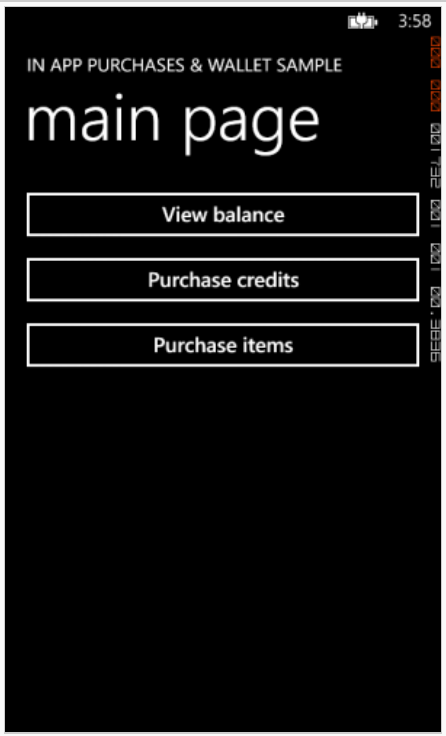
Go back to the **Apps** section of the Developer Center, select the application you created, navigate to the **Details** tab and copy the value of the **App ID** field. We are going to need it in the next step.

Adding in-app purchase functionality

Now that we have everything configured and running in the backend, we can proceed to integrate the items with our application.

Obtaining purchasable items and creating the application's Wallet

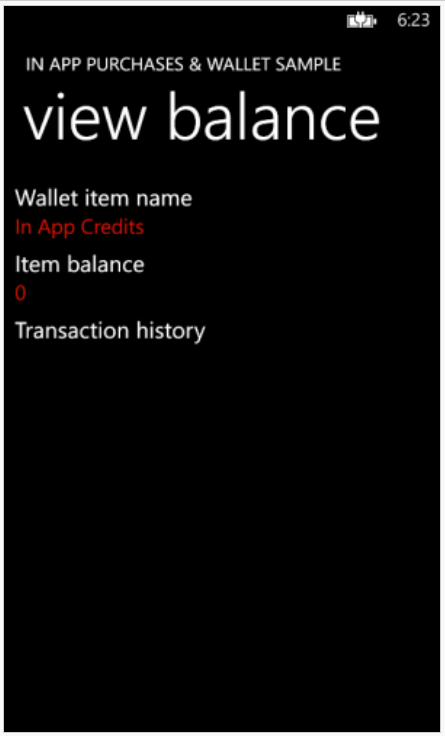
First, go back to editing the **WMAppManifest.xml** and under the **Packaging** tab, paste the GUID you obtained from uploading your beta app in the **Product ID** field (remember that it must be surrounded in braces). By doing this, the Dev Center backend will identify our app when making any request and returning the appropriate purchasable products when querying for them. Now we are ready to start editing the layout of the application's pages.



Main application page.

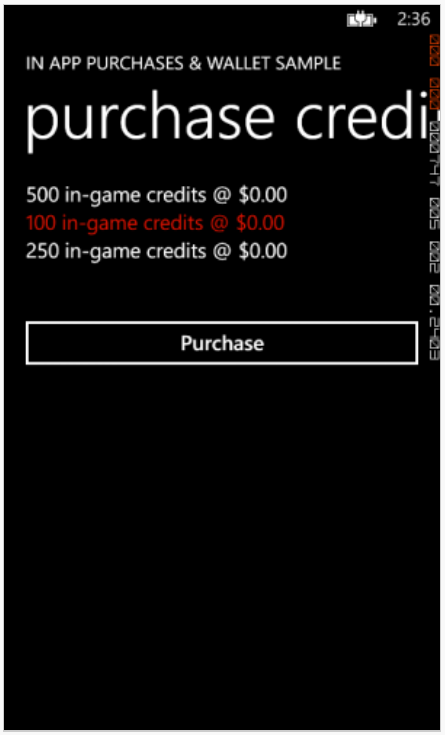
This is the layout of the main page. Each button navigates to the corresponding page, and the user will be able to select what action he wants to do from there: check current balance, purchase more credits through in-app purchases or purchase fictitious in-game items.

(C) Copyright Nokia 2013. All rights reserved.



View balance page.

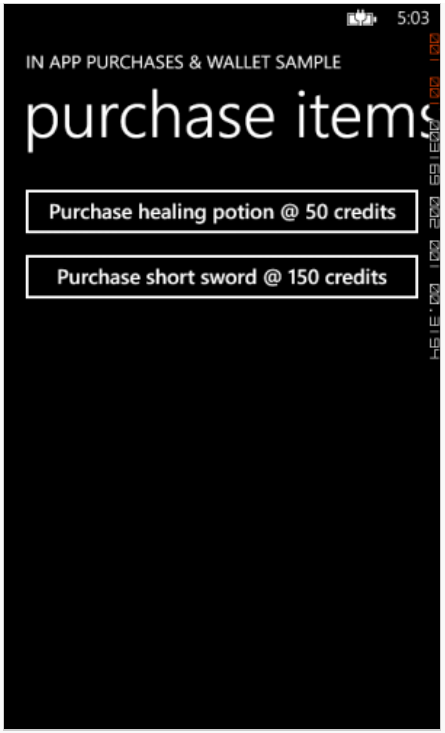
In the **ViewBalance.xaml** page we can see the name we gave to our in-game currency, the current balance (initially 0) and a list containing the transaction history (initially empty).



Purchase credits page.

In **PurchaseCredits.xaml** we can select a specific in-app item to purchase for real money. As you can see, the list shows the items we previously created in Dev Center listed at $0.00, since we are running a beta app.

(C) Copyright Nokia 2013. All rights reserved.



Purchase items page.

Finally, the **PurchaseItems.xaml** page allows us to do a mock purchase of items usable in our game, showing their name and price in in-game credits.

For simplicity's sake, we are going to base our viewmodel on GalaSoft's MVVM Light Toolkit, so download it and add the proper references to your project or get it from NuGet. Let's specify what we will need in our viewmodel:

- `WalletTransactionItem walletCredits` (private field): this will be a handle to the item stored in the Wallet, like currency name, amount and transaction history.
- `int DisplayBalance` (public property): gets and set the amount of credits stored in the **WalletTransactionItem**.
- `string DisplayName` (public property): gets the name of the currency.
- `ObservableCollection<WalletTransaction> TransactionHistory` (public property): gets a collection of **WalletTransaction** that maps all currency balance changes.
- `ObservableCollection<ProductListing> InAppProducts` (public property): stores the purchasable items obtained by querying Dev Center.

Now let's explain the constructor:

```
public MainViewModel()
{
    walletCredits = null;
    InAppProducts = new ObservableCollection<ProductListing>();

    if (!IsInDesignMode)
    {
        CreateWalletItem();
        LoadInAppProducts();
    }
}
```

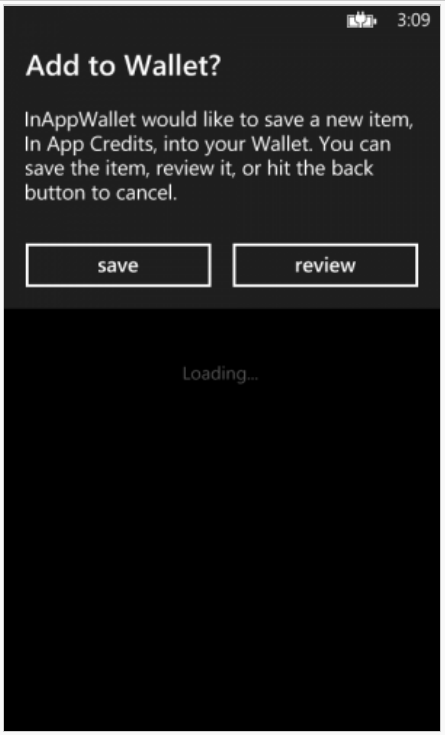
Pretty straightforward, apart from the default variable initialization we relegate the Wallet item creation and in-app querying to their own functions.

Warning: In the following code we have created an extension method `BitmapImage.LoadFromContent` so the images are immediately loaded. Failing to do so or not specifying any of the `Logo` parameters will throw an exception when creating the `WalletTransactionItem`.

```
private void CreateWalletItem()
{
    WalletTransactionItem item = Wallet.FindItem(WalletItemId) as WalletTransactionItem;

    if (item == null)
    {
        BitmapImage logo99 = new BitmapImage();
        logo99.LoadFromContent(new Uri("Assets/Wallet/Logo99x99.png", UriKind.Relative));
        BitmapImage logo159 = new BitmapImage();
        logo159.LoadFromContent(new Uri("Assets/Wallet/Logo159x159.png", UriKind.Relative));
        BitmapImage logo336 = new BitmapImage();
        logo336.LoadFromContent(new Uri("Assets/Wallet/Logo336x336.png", UriKind.Relative));

        item = new WalletTransactionItem(WalletItemId)
        {
            DisplayName = "In App Credits",
            Logo99x99 = logo99,
            Logo159x159 = logo159,
            Logo336x336 = logo336,
            DisplayBalance = DefaultBalance.ToString(),
            TransactionHistoryNavigationUri = new Uri("/View/ViewBalance.xaml", UriKind.Relative),
        };

        AddWalletItemTask addItemTask = new AddWalletItemTask();
        addItemTask.Item = item;

        addItemTask.Show();
    }

    walletCredits = item;
}
```


(C) Copyright Nokia 2013. All rights reserved.

We ask the Wallet if an item with the unique ID specified in `WalletItemId` already exists; if not, we create a new `WalletTransactionItem` with an initial balance of 0 and ask the user to accept its creation by calling `AddWalletItemTask.Show`. Finally, we store its value in our local `walletCredits` handle.



Asking the user to add the new item to the Wallet.

```
private async void LoadInAppProducts()
{
    try
    {
        ListingInformation productList = await Windows.ApplicationModel.Store.CurrentApp.LoadListingInformationAsync();

        foreach (KeyValuePair<string, ProductListing> product in productList.ProductListings)
        {
            InAppProducts.Add(product.Value);
        }
    }
    catch (Exception e)
    {
        System.Windows.MessageBox.Show("Couldn't retrieve purchasable items.");
    }
}
```

This is the code for the function that loads the in-app products. It's marked as `async` because the call to `LoadListingInformationAsync` must be awaited and can take some time depending on network conditions, so it's better to call it without blocking the rest of the application workflow. If Dev Center's infrastructure returns any results we store their information locally; if not, we catch the exception and tell the user that something went wrong.

Purchasing credits

Create the **PurchaseCredits.xaml** page and add a `ListBox` and a `Button` to it. Databind the `ListBox.ItemsSource` property to the `InAppProducts` property of the viewmodel, so the UI properly reflects the items we have available to purchase. Finally, add a new callback to the `Click` event of the button to enable the purchasing code:

```
private void Purchase_Click(object sender, System.Windows.RoutedEventArgs e)
{
    if (InAppList.SelectedItem != null)
    {
        viewModel.PurchaseInAppItem(InAppList.SelectedItem as ProductListing);
    }
}
```

And move back to **MainViewModel.cs** to implement the `PurchaseInAppItem` function we just used. Here is a brief explaination of what it is going to perform:

- Tell the backend that we are going to purchase an item by calling `CurrentApp.RequestProductPurchaseAsync`; the first parameter is the product ID and the second specifies if we want a purchase receipt or not. This function returns a string containing a XML-serialized receipt that will allow you to verify if your purchase was legit or someone tampered with it.

> **Note:** Receipt requesting only works if your app isn't in beta mode. You can find more info about the receipt's content and how to validate its authencity in this MSDN article.

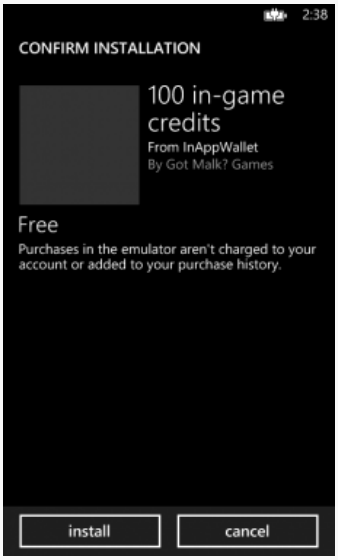
- Check if the `ProductId` matches any of the IDs we added in Dev Center, and store the appropriate credits value in a temporal variable. If not, show an error message; in you application, this could be an indicator of an expired offer or another type of error.
- Create a new `WalletTransaction` and fill it with the details of the transaction. In beta mode you must fill it manually, but if you have a valid receipt you can retrieve this information from it.
- Increment the amount of credits available in the Wallet.
- Finally, add the transaction to the `TransactionHistory` and call `ReportProductFulfillment` to tell the backend that our item purchase was successful.

(C) Copyright Nokia 2013. All rights reserved.

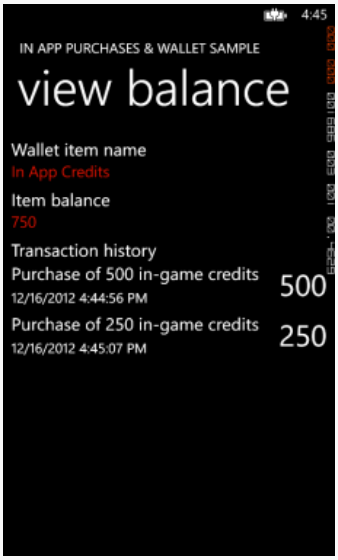
Make sure you call this when a consumable item has been purchased, otherwise you won't be able to re-buy it due to a protection against accidental double purchases.

- Tell the `WalletTransactionItem` to save the changes.

```
public async void PurchaseInAppItem(ProductListing product)
{
    string receipt = await Windows.ApplicationModel.Store.CurrentApp.RequestProductPurchaseAsync(product.ProductId, true);
    int creditsAmount = 0;

    switch (product.ProductId)
    {
        case InApp100CreditsID:
            creditsAmount = 100;
            break;
        case InApp250CreditsID:
            creditsAmount = 250;
            break;
        case InApp500CreditsID:
            creditsAmount = 500;
            break;
        default:
            MessageBox.Show("Unknown product ID: " + product.ProductId);
            return;
    }

    WalletTransaction transaction = new WalletTransaction();
    transaction.TransactionDate = DateTime.Now;
    transaction.Description = "Purchase of " + product.Name;
    transaction.DisplayAmount = creditsAmount.ToString();

    int currentBalance = int.Parse(walletCredits.DisplayBalance);
    currentBalance += creditsAmount;
    walletCredits.DisplayBalance = currentBalance.ToString();

    walletCredits.TransactionHistory.Add(new KeyValuePair<string, WalletTransaction>(Guid.NewGuid().ToString(), transaction));
    Windows.ApplicationModel.Store.CurrentApp.ReportProductFulfillment(product.ProductId);
    await walletCredits.SaveAsync();
}
```



Page confirming the item purchase.



Balance after purchasing credits.

Purchasing items with the in-game credits

Lastly, create the **PurchaseItems.xaml** page and add two buttons to them. One will allow us to buy a health potion and the other will purchase a short sword. As before, add a new callback to their `Click` events:

```
private void PurchasePotion_Click(object sender, System.Windows.RoutedEventArgs e)
{
    viewModel.PurchaseItemWithCredits("Health potion", 50);
}

private void PurchaseSword_Click(object sender, System.Windows.RoutedEventArgs e)
{
    viewModel.PurchaseItemWithCredits("Short sword", 150);
}
```

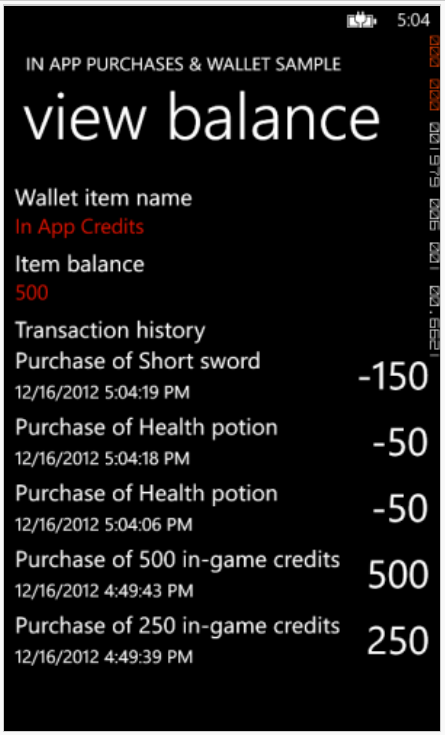
And proceed to implement the `PurchaseInAppItem` function in the viewmodel:

- First, we check if we have enough balance to purchase the item. If not, display an error message.
- Create a new transaction reflecting the purchase of the item.
- Decrement the current balance of the `WalletTransactionItem`.
- Add the transaction to the `TransactionHistory` and save the changes.

```
public async void PurchaseItemWithCredits(string itemName, int price)
{
    int currentBalance = int.Parse(walletCredits.DisplayBalance);

    if (currentBalance < price)
    {
```


(C) Copyright Nokia 2013. All rights reserved.

```
        MessageBox.Show("You don't have enough credits to purchase " + itemName);
    }

    WalletTransaction transaction = new WalletTransaction();
    transaction.TransactionDate = DateTime.Now;
    transaction.Description = "Purchase of " + itemName;
    transaction.DisplayAmount = (-price).ToString();

    currentBalance -= price;
    walletCredits.DisplayBalance = currentBalance.ToString();

    walletCredits.TransactionHistory.Add(new KeyValuePair<string, WalletTransaction>(Guid.NewGuid().ToString(), transaction));
    await walletCredits.SaveAsync();
}
```



Balance page showing the purchase of the in-game items.

And this is the basic functionality of both in-app purchases and Wallet. Feel free to modify and test the complete source code to suit your needs: Media:InAppWallet.zip.

Remarks

Please note that this article is only a proof-of-concept of both the in-app purchase system and Wallet introduced in Windows Phone 8. If you are going to use this code as a base for your own projects, make sure you implement correct security mechanisms to control errors while the item is bought and stored. A good starting point would be creating your own backend so you store the receipts remotely, so if a user deletes the application or migrates to a new phone he doesn't lose all his consumable purchases.

(C) Copyright Nokia 2013. All rights reserved.

(C) Copyright Nokia 2013. All rights reserved.