



P r o f e s s i o n a l E x p e r t i s e D i s t i l l e d

Learning Windows Azure Mobile Services for Windows 8 and Windows Phone 8

A short, fast and focused guide to enhance your Windows 8 applications by leveraging the power of Windows Azure Mobile Services

Geoff Webber-Cross

www.allitebooks.com

[PACKT] enterprise 
PUBLISHING professional expertise distilled

Learning Windows Azure Mobile Services for Windows 8 and Windows Phone 8

A short, fast and focused guide to enhance your
Windows 8 applications by leveraging the power of
Windows Azure Mobile Services

Geoff Webber-Cross



BIRMINGHAM - MUMBAI

Learning Windows Azure Mobile Services for Windows 8 and Windows Phone 8

Copyright © 2014 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: January 2014

Production Reference: 1090114

Published by Packt Publishing Ltd.

Livery Place
35 Livery Street
Birmingham B3 2PB, UK.
ISBN 978-1-78217-192-8

www.packtpub.com

Cover Image by Artie Ng (artherng@yahoo.com.au)

Credits

Author

Geoff Webber-Cross

Project Coordinator

Michelle Quadros

Reviewers

Aidan Casey

Rafał Jońca

Peter Kirchner

Proofreader

Mario Cecere

Indexer

Rekha Nair

Acquisition Editor

Neha Nagwekar

Grant Mizen

Graphics

Yuvraj Mannari

Lead Technical Editor

Madhujā Chaudhari

Production Coordinator

Nilesh Bambardekar

Technical Editors

Kapil Hemnani

Mrunmayee Patil

Cover Work

Nilesh Bambardekar

Copy Editors

Alisha Aranha

Roshni Banerjee

About the Author

Geoff Webber-Cross has commercial and personal experience of developing Windows 8 and Windows Phone applications and using Azure for websites, mobile services, web services, and Windows services. He enjoys learning about new technologies and solving difficult software problems.

I'd like to thank my wife for putting up with me tapping away on my laptop every night for months on end while writing this book.

About the Reviewers

Aidan Casey has over 16 years' experience in the software industry. He lives in Ireland and works remotely as a solution architect for MYOB, Australia. He is a passionate member of the technical community and a regular presenter at events across Australia and Ireland. Outside of work, he enjoys running barefoot and solving world's problems over a pint of Guinness!

Rafał Jońca has over 10 years of web development experience. In the past, he was a lead developer responsible for creating high traffic websites (in PHP, Python, and JavaScript), game servers (in node.js), and Smart TV in-house framework (in JavaScript). Currently, he is the owner of Gluwer – a small company where he works as an independent consultant. He helps his clients in topics related to web services and website development using node.js and the Windows Azure cloud. Also, he has over 13 years of experience in translating over 40 IT books about Flash, PHP, Java, JavaScript, agile, and SQL into Polish.

Peter Kirchner has worked as a technical evangelist at Microsoft Germany since 2008. In this role, he speaks at conferences and writes articles that focus on cloud computing, with the goal to inspire new technologies. Also, he supports developers and administrators to develop and use the Microsoft platform. Before working at Microsoft, he gained experience while working in the area of SharePoint development and consulting. As a student, he showed great interest in network technologies, security, and distributed systems, and he graduated with a diploma in Computer Science.

I am very grateful to my employer for encouraging my passion for technology and my fiancé Lena for her everlasting patience.

www.PacktPub.com

Support files, eBooks, discount offers, and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<http://PacktLib.PacktPub.com>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Instant Updates on New Packt Books

Get notified! Find out when new books are published by following [@PacktEnterprise](#) on Twitter, or the *Packt Enterprise* Facebook page.

Table of Contents

Preface	1
Chapter 1: Preparing the Windows Azure Mobile Services Portal	7
Choosing a subscription	7
Pay-as-you-go subscription	8
Basic and Standard subscriptions	8
Free trial	8
Creating a Windows Azure account	9
Creating a mobile service	10
Mobile Services features	13
Managing keys	13
Mobile service dashboard	14
Configure	15
Scale	16
Logs	18
Summary	18
Chapter 2: Start Developing with Windows Azure Mobile Services	19
Preparing our development environment	19
Requirement for hardware	19
Setting up the software	20
Requirement for store accounts	20
Creating apps from the portal	21
Connecting existing apps to Windows Azure Mobile Services	23
Adding a Connected Service in Visual Studio 2013	24
Manually installing the SDK in Visual Studio 2012 Express for Windows Phone	25
Creating a table	25
Writing a model of the table	26

Interacting with the table	27
Summary	30
Chapter 3: Securing Data and Protecting the User	31
Configuring permissions	32
Rules for choosing permissions	32
Authentication providers	33
Authentication	33
Registering for Windows Live Connect Single Sign-on	34
Authentication in the app	35
Logging in	35
Storing credentials	37
Logging out	39
The DataServiceBase class	40
REST API and the master key	43
Summary	45
Chapter 4: Service Customization with Scripts	47
Understanding table scripts	47
Level-insert table script example	48
Score-insert script example	49
Score-read script example	49
API scripts	50
Creating an API script	50
High-score API script	51
Script debugging and logs	54
Scheduling	54
Working locally with Git	56
Pulling the repository	56
Updating our repository	57
Adding scripts manually	57
Pushing back changes	57
Implementing NPM modules	58
Summary	59
Chapter 5: Implementing Push Notifications	61
Understanding Push Notification	
Service flow	62
Setting up Windows Store apps	63
Setting up tiles	66
Setting up badges	66
Setting up Windows Phone 8 apps	68
Service scripts	72

WNS scripts for Store apps	74
Sending toast notifications	75
Sending tile notifications	75
Sending multiple tiles	76
Sending badge notifications	77
MPNS scripts for Windows Phone apps	77
Sending toast notifications	77
Sending tile notifications	78
Summary	79
Chapter 6: Scaling Up with the Notifications Hub	81
Configuring the Hub	82
Setting up Windows Store and Windows Phone 8 apps	84
Calling the hub from scripts	86
Creating WNS scripts (for Store apps)	87
Sending toast notifications	87
Sending tile notifications	87
Sending badge notifications	88
Creating MPNS scripts (for Windows Phone 8 apps)	88
Sending toast notifications	89
Sending tile notifications	89
Backend services	90
Targeting audience using tags	91
Summary	93
Chapter 7: Best Practices for Web-connected Apps	95
App certification requirements for the Windows Store	95
UX guidelines	96
Implementing a privacy policy	98
Checking the network connection	98
Managing notifications settings	100
Implementing settings pages	103
Summary	104
Index	107

Preface

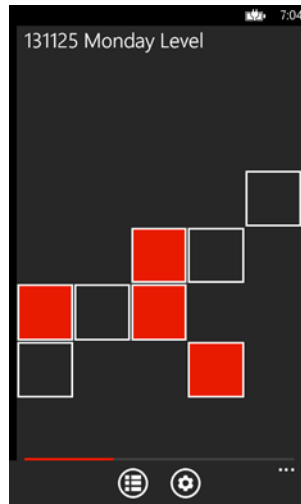
Windows Azure offers a wide range of cloud-based services, which are hosted on a robust, well-managed infrastructure, and can be easily scaled to meet our business demands. Windows Azure Mobile Services is a fantastic member of the Azure family, which allows mobile developers to quickly build web-connected applications and enhance user experience with push notifications.

Using traditional web technology, we will need to think about creating databases and web services, deciding what security mechanisms to use; build tools to administer the data and services; and write backend services to interface with the different Push Notification Service providers we want to use.

With Windows Azure Mobile Services, we can build model-first services without touching the database schema, get a fully managed and REST API for our data without writing a line of code, and modify the database API methods using scripts. Using scripts, we can also create API methods to access data, send push notifications, and make HTTP requests.

This book aims to investigate all that Windows Azure Mobile Services has to offer with practical examples, which can be used in real applications. Also, it covers areas of application development to enhance user experience, help with store certification, and improve development efficiency. I've created a simple game named TileTapper in C#/XAML for Windows 8 and Windows Phone to help illustrate use cases for all the service features and keep the book real!

The TileTapper game consists of a grid board seeded from a simple Boolean array of active or inactive tiles. When the app launches, it prompts the user to log in using the Windows Live connect authentication provider, downloads levels and current high score from our backend service, and then begins the game. The phone app game grid looks like the following screenshot:



The user has to tap on all the tiles before the time runs out to complete the level. The score at the end of the game as well as high score are stored in the service, if needed. Levels are generated automatically using a scheduled script and push notifications are sent about new high scores achieved and new levels created. Both apps have settings pages for managing notifications.

What this book covers

Chapter 1, Preparing the Windows Azure Mobile Services Portal, explains how to choose a subscription, set up an Azure account, and create a Mobile Service; it also talks about the current Mobile Services portal features.

Chapter 2, Start Developing with Windows Azure Mobile Services, covers what software and hardware you need to develop Windows 8 and Windows Phone 8 apps using Windows Azure Mobile Services. We'll also learn about creating preconfigured apps from the portal and connecting existing apps from scratch.

Chapter 3, Securing Data and Protecting the User, looks at permission options for tables and APIs and different authentication methods for protecting our data and users' personal information. We'll also look at developing code to log in users with an authentication provider and storing their credentials for subsequent app usage.

Chapter 4, Service Customization with Scripts, covers customizing scripts to perform validation, manipulate data, and make HTTP requests. We'll also look at installing a Node npm package and using it in our scripts and finally, using the Git version control to pull a copy of our scripts to work locally and as a backup.

Chapter 5, Implementing Push Notifications, explains configuring Windows Store and Windows Phone 8 apps to implement push notifications; create a channels table to manage push channel URIs; and send Tile, Toast, and Badge notifications using the MPNS (Windows Phone) and WNS (Windows Store) providers.

Chapter 6, Scaling Up with the Notifications Hub, looks at the benefits of using the Notifications Hub from the service bus family of services, building on *Chapter 5, Implementing Push Notifications*. We adapt our code to register the push channel URI with the Notifications Hub, create scripts for sending notifications using the Azure for Node SDK, and use the Windows Azure Service Bus SDK to send notifications from .NET backend services.

Chapter 7, Best Practices for Web-connected Apps, looks at what we need to do to get our apps certified with respect to our Windows Azure Mobile Services implementation. We'll look at the app certification requirements for the Windows Store and UX guidelines and then talk about privacy statements, checking the cost impact of using the Internet connection and managing push notifications.

What you need for this book

Chapter 2, Start Developing with Windows Azure Mobile Services, details what software and hardware is needed, but as an overview, you need a machine with Windows 8.1 installed. If you want to create a phone app, it needs to be capable of running the Windows Phone 8 Hyper-V emulators. Visual Studio Express 2013 for Windows is needed for Windows 8 Store apps and Visual Studio Express 2012 for Windows Phone for phone apps. When we look at managing scripts with the Git version control, we need Git and also node.js for installing NPM modules.

Who this book is for

This book is aimed at developers wishing to build Windows 8 and Phone 8 applications with Windows Azure Mobile Services implementation. Basic C# and JavaScript skills are advantageous; also some knowledge of building Windows 8 or Windows Phone 8 applications is required.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows "The MobileServiceClient class has a Logout method that doesn't seem to do anything other than clear the `CurrentUser` property."

A block of code is set as follows:

```
public void Logout() {  
    this._mobileService.Logout();  
  
    // Clear credentials  
    StorageHelper.StoreSetting(USER_ID, null, true);  
    StorageHelper.StoreSetting(USER_TOKEN, null, true);  
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
GET https://tiletapper.azure-mobile.net/tables/  
leaderboard HTTP/1.1  
X-ZUMO-APPLICATION: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX  
Host: tiletapper.azure-mobile.net
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Once this is done, copy **Client ID** and **Client secret (v1)** to the **microsoft account settings** section on the **IDENTITY** tab in the Windows Azure Mobile Services portal and click on **SAVE**."

 Warnings or important notes appear in a box like this.

 Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book – what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to feedback@packtpub.com, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books – maybe a mistake in the text or the code – we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from <http://www.packtpub.com/support>.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at questions@packtpub.com if you are having a problem with any aspect of the book, and we will do our best to address it.

1

Preparing the Windows Azure Mobile Services Portal

Before we get down to any coding or even looking at development tools, we need to do some work on getting things prepared in Windows Azure. In this chapter, we're going to talk about the following:

- Choosing a pricing plan for services you wish to implement
- Creating a Windows Azure account that allows you to use any Windows Azure services
- Creating our first mobile service
- Exploring the Mobile Service portal

To use Windows Azure Services and create application store accounts, you're going to need a Microsoft account (formerly known as Microsoft Live ID). If you haven't already got one, go and create one here <https://signup.live.com/signup.aspx>.

Choosing a subscription

To get started, go to <http://www.windowsazure.com> and first check out the pricing options; there will be a **PRICING** tab and a **Mobile Services** option under the **COMPUTE** header. Take a look at the pricing calculator for **mobile services** at <http://www.windowsazure.com/en-us/pricing/calculator/?scenario=mobile> and have a quick look to make sure you have an idea of how much the services you want to use might cost. If you don't know what you want, just sign up for a free account.

Pay-as-you-go subscription

Small apps and a bit of experimentation are unlikely to cost you anything. At the time of writing this, you get the following for free. But check for yourself so that you're not in for a nasty surprise if you sign up for a **Pay-as-you-go (PAYG)** account and exceed your usage:

- 10 Mobile Services
- 20 MB SQL database for 12 months
- 500 K API calls per month
- Send push notifications via the Notification Hubs to up to 500 active devices
- 1,00,000 Notification Hubs operations per month

Throughout the book, I'll try to point out where you need to be careful to make sure you don't start incurring costs if you want to maintain free service usage.

Basic and Standard subscriptions

Basic and **Standard** subscriptions need you to buy units (service instances) for the number of API calls you expect to make. If you can calculate how many API calls your apps are likely to be making and how much storage you need, you can decide if either of these subscriptions will be the most economical for you.

Free trial

The free trial allows you to use 200 USD worth of any services (not just mobile) you like per month.

Creating a Windows Azure account

If you already have a Windows Azure account, skip to the next section; otherwise, click on the **Portal** tab (<https://manage.windowsazure.com/>). It will take you to log in using your Microsoft account if you are not already logged in. Once you have logged in, you will see a page saying you have no subscription. Click on the **SIGN UP FOR WINDOWS AZURE** link, <https://account.windowsazure.com/SignUp>. You should end up on the **Sign up** page (There are a number of routes to get to this page through the website, but this seemed to be the least clicks for me!). Your personal details should appear on your details in the account info page and you'll need to verify it with an SMS message or a call verification:

Sign up

Free Trial
Learn more ▾

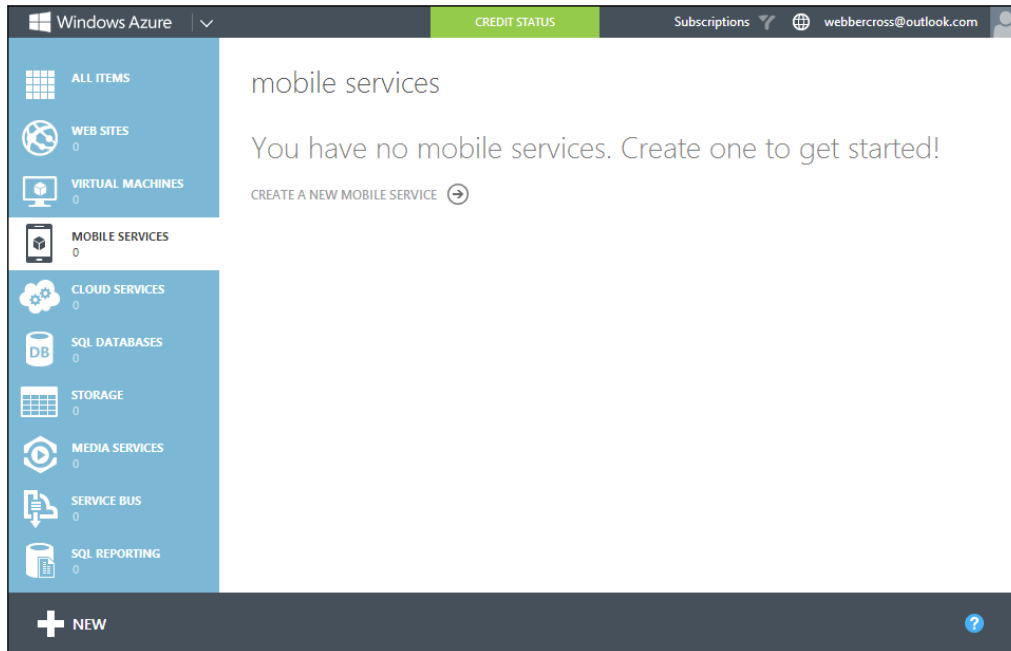
Windows Azure geoff@outlook.com ▾

- 1 About you
FIRST NAME LAST NAME COUNTRY/REGION
Geoff Webber-Cross United Kingdom ▾
VATID CONTACT EMAIL COMPANY NAME
Optional geoff@outlook.com Optional
- 2 Mobile verification
☒ Send text message ☐ Call me
United Kingdom (+44) ▾
Send text message
Verify code
- 3 Payment information
- 4 Agreement
☐ I agree to the Windows Azure Agreement, Offer Details, and Privacy Statement.
☐ Microsoft may use my email and phone to provide special Windows Azure offers.

Once verified, you can enter your credit card details. You can also sign up for a free trial or a pay-as-you-go account. Don't panic, you don't get automatically signed up for any premium subscriptions; however, 1 USD will be charged to your credit card for verification. Accept the agreement and click on the **Purchase** button, your card details will be validated and you will be taken to the subscriptions page where you'll be pleased to find you already have a free trial! From here, you can add subscriptions to meet your own requirements. If you have chosen a trial subscription, there is a spending limit feature so you don't incur any costs; once you reach the offer limits, services will be disabled and data will be available as read only.

Creating a mobile service

Now we've got all the boring sign up stuff out of the way, we can get to the bit we're interested in. Go to the portal (<https://manage.windowsazure.com>) and it's probably a good idea to bookmark the page in your browser as we'll be here quite a bit. The portal should look something like the following screenshot, displaying all the Windows Azure services on the left available to us:



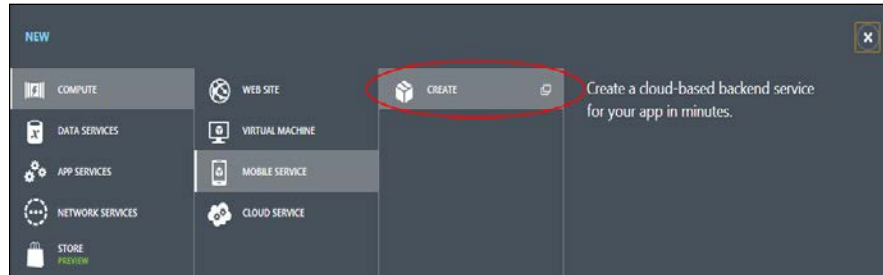
The Windows Azure portal offers a plethora of services, but we're obviously going to concentrate on Windows Azure Mobile Services and will touch upon Windows Azure SQL Databases and Windows Azure Service Bus when we look at the Notification Hubs.

To create a new mobile service perform the following steps:

1. Click on the **+ NEW** toolbar button shown as follows:



2. Select **CREATE** from the pop-up menu shown as follows:



3. Fill in the details for the service. I'm going to opt to use my PAYG subscription, **Create a free 20MB SQL database**, and target **North Europe**.

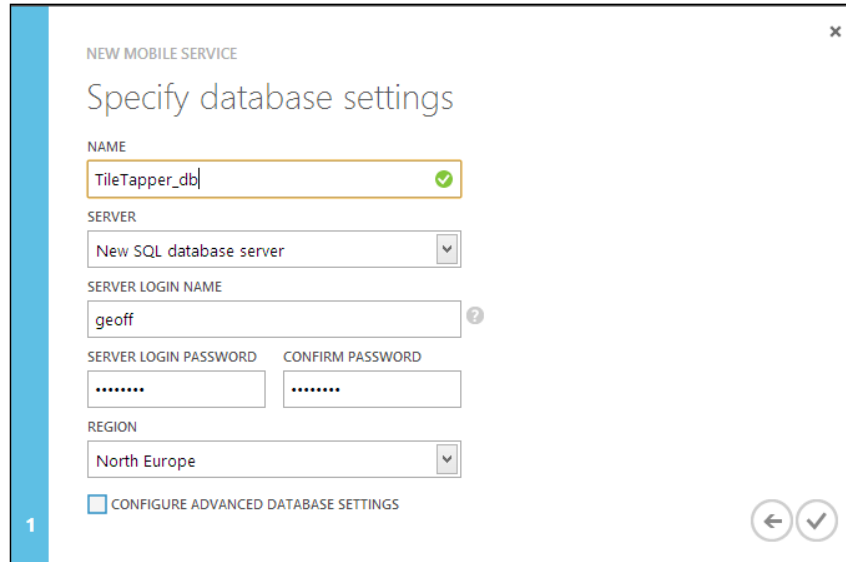
A screenshot of the 'NEW MOBILE SERVICE' form in the Azure portal. The form is titled 'Create a Mobile Service'. It contains the following fields:

- URL**: A text input field containing 'TileTapper'.
- DATABASE**: A dropdown menu with the selected option 'Create a free 20 MB SQL database'.
- SUBSCRIPTION**: A dropdown menu with the selected option 'Pay-As-You-Go'.
- REGION**: A dropdown menu with the selected option 'North Europe'.

At the bottom right of the form, there is a blue bar with a right-pointing arrow and the number '2'.

At this point, if we choose the **Create a new SQL database** instance, we will start incurring costs for the new database. If we had already created a database, we would see this as an option too. Choose a region close to where your target audience is likely to be so that the service is hosted as close to them as possible. Mobile Services does not use affinity groups, so you have to specify a region.

4. The next screen will show us options for creating a database instance:



The screenshot shows a web form titled "NEW MOBILE SERVICE" with a sub-header "Specify database settings". The form contains several input fields: "NAME" with the value "TileTapper_db" and a green checkmark; "SERVER" with a dropdown menu showing "New SQL database server"; "SERVER LOGIN NAME" with the value "geoff" and a help icon; "SERVER LOGIN PASSWORD" and "CONFIRM PASSWORD" fields with masked text "*****"; and "REGION" with a dropdown menu showing "North Europe". At the bottom left, there is a checkbox labeled "CONFIGURE ADVANCED DATABASE SETTINGS" which is currently unchecked. At the bottom right, there are two circular buttons: a back arrow and a checkmark. A blue vertical bar on the left side of the form contains the number "1".

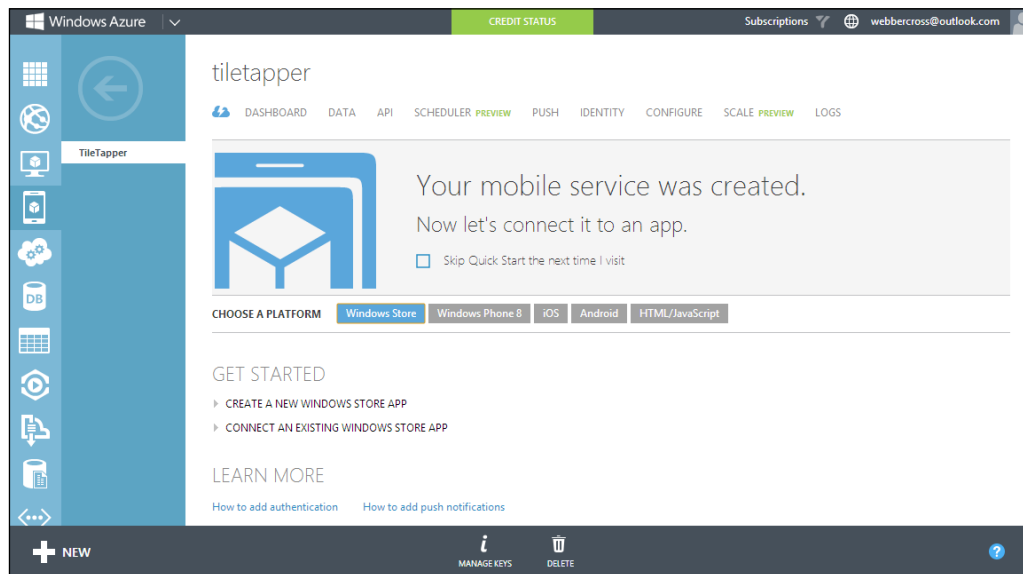
At this point, we need to choose a name for the database, set the login credentials (make a note of them for future reference), and choose a location for the database server. By default, the mobile service with the suffix **_db** is set as the database name; this is fine for me as I only want to use it for one service. However, if you don't want to spend money on more databases and want to use it for multiple applications, you may want to choose a more generic database name, something like `AppsDatabase`. It is sensible to host the database server at the same location as your mobile service instance, so that additional transfer costs are not incurred and they don't have to talk to each other across the world every time a request is made!

5. I'm going to choose default database settings, but you can check **CONFIGURE ADVANCED DATABASE SETTINGS** and you will be able to change the collation of the database.

This page actually displays a message stating that we won't be charged for the database configuration we've chosen, But if you change the database size, it will become a paid database.

Mobile Services features

Now that we've created a mobile service, we can explore the features available to us in the portal. From the main portal, select **Mobile Services** and then click on the service you have just created to navigate to the Mobile Services portal:



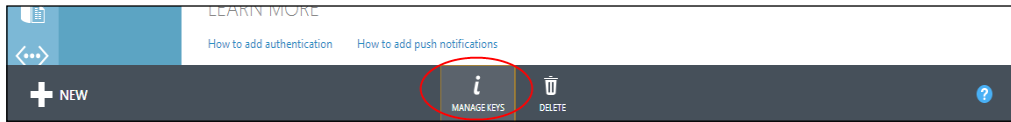
Along the top are all features available to us to help build our services and applications. The bottom toolbar is context sensitive and has actions for the selected feature.

At the time of writing this, a number of features had a **PREVIEW** tag next to them; you may also see beta and prerelease features. These features are likely to become fully supported. However, if you use them, you need to bear in mind that they may be changed or be completely removed. There is a terms of use article here, which is worth a read: <http://www.windowsazure.com/en-us/support/legal/preview-supplemental-terms/>. We'll investigate all the features, even the preview ones just for completion. When you are reading this, there are likely to be more features.

Managing keys

Windows Azure Mobile Services have an **application key** and a **master key**, which limit access to the API. Tables and APIs can be set to only grant access to calls from application requests bearing the application key embedded in the application code. However, it is not encrypted so is not considered secure. This means it is important to authenticate users before accessing services.

The master key is used for administrator access and should not be distributed with the app. These keys can be managed from the **MANAGE KEYS** button on the bottom toolbar, which appears on the main portal and various tabs:

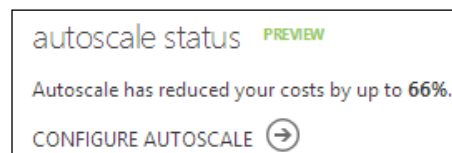


The keys can be regenerated if there has been a security compromise, but should not be changed unless absolutely necessary as it will stop all published apps from accessing services and will mean they need to be republished with the new key.

Mobile service dashboard

This is an overview of what's going on with our service. The top section displays a chart of our API and data usage; there are filters to change the time period and y-axis scaling. The dashboard displays the following sections:

- **Mobile service endpoint status:** This allows us to monitor the availability of our service (this is only available for premium subscriptions) when endpoint monitoring has been configured. If you have a critical system, this is an important feature for making sure the service is meeting your SLA.
- **Usage overview:** This is an overview of your API call, active device, and out data usage.
- **Autoscale status:** If you have scaling enabled, this displays the current scaling status. This can be set up by clicking on the **CONFIGURE AUTO SCALE** link or going to the **Scale** tab. When enabled, the dashboard tells us about how much cost reducing scaling is being achieved (depending on the demand):



This is a round about way of saying we've got one of three possible instances running.

- **Quick glance:** This section on the right and has a quick summary of the service's current status.

- **Data:** The **Data** tab lists all the tables configured in our database, shows us an overview, and allows us to browse the data, modify the operation scripts, edit columns, and change the permissions. These features will be discussed in detail in subsequent chapters.
- **API:** The **API** tab allows us to manage custom APIs implemented in our service. Each table has a default set of operation scripts that can be modified. APIs allow us to create any operation that can make HTTP requests and perform database operations. Each API has a standard set of HTTP methods that can be implemented as required.
- **Scheduler:** From here, we can create and manage scheduled jobs that can run scripts on a timed schedule or on demand.
- **Push:** For me, this is one of the coolest features of Windows Azure Mobile Services that allows us to manage push notifications to our applications, without having to manually create and host our own services, which interface with **Windows Push Notification Services (WNS)**, **Apple Push Notification Service (APNs)**, and **Google Cloud Messaging (GCM)** service. We'll also look at the Notification Hubs, which is a more scalable way of achieving push notifications; however, it's not configured directly from the Mobile Services portal.
- **Identity:** Windows Azure Mobile Services delegates its authentication to providers such as Microsoft account, Facebook, Twitter, and Google. This means we don't need to worry about storing and managing user credentials or manually dealing with authentication mechanisms such as OAuth2. This tab is where we configure the identity provider used to authenticate our application.

Configure

The **Configure** tab contains miscellaneous settings for Windows Azure Mobile Services as follows:

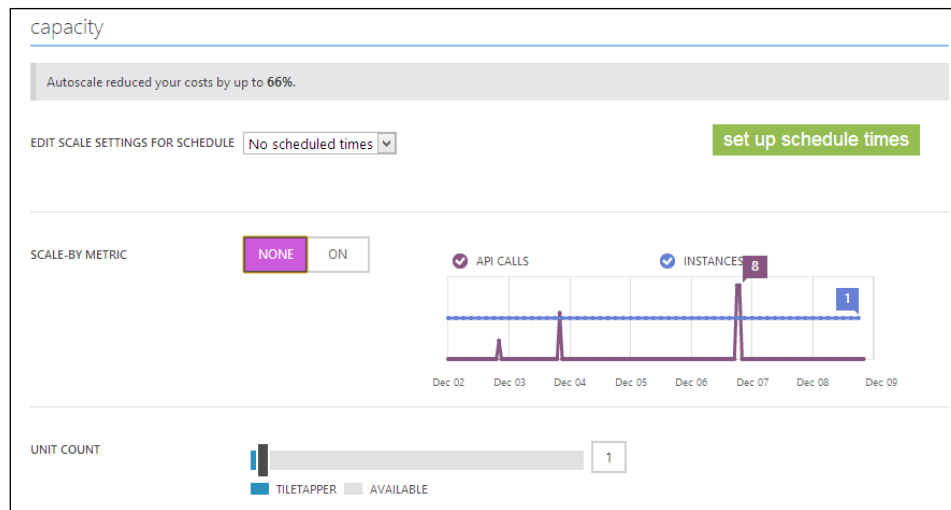
- **Database settings:** This section contains two links for configuring the database and database server that Mobile Services use. Both of these links take us out of the Mobile Services portal and into the SQL Databases portal.
- **Source control:** It's possible to manage the scripts used by the service (we'll discuss these later in the book) using Git version control, by initially pulling the repository to your machine, working locally, and then pushing back updates you have made, instead of working in the portal. Once this is set up, the dashboard displays the source control username.

- **Dynamic schema:** This setting allows you to enable or disable the **Dynamic Schema** feature. The feature allows the service to automatically add columns to tables as they appear through the API so that you don't have to constantly modify your database schema while you develop your services. It is recommended that this feature is disabled once development is finished and your app is in production.
- **Cross-origin resource sharing (CORS):** This section allows you to create a list of hosts that are permitted to interact with your mobile service (including wildcards such as `*.example.com`). Client-side JavaScript originating from hosts in the list will be granted access to the service, otherwise they will be denied. This does not affect native apps using the APIs.
- **Developer analytics:** This section allows you to set up the application performance analytics.
- **App settings:** These are key-value pair values you can use and access in scripts to help with things such as string settings, which you may want to change from the dashboard rather than in the script. This is similar to the `AppSettings` section in `web.config` and `app.config` files.
- **Monitoring:** If you have a premium subscription, up to two monitoring endpoints can be configured from here, allowing you to monitor the service availability from up to three geo-distributed locations.

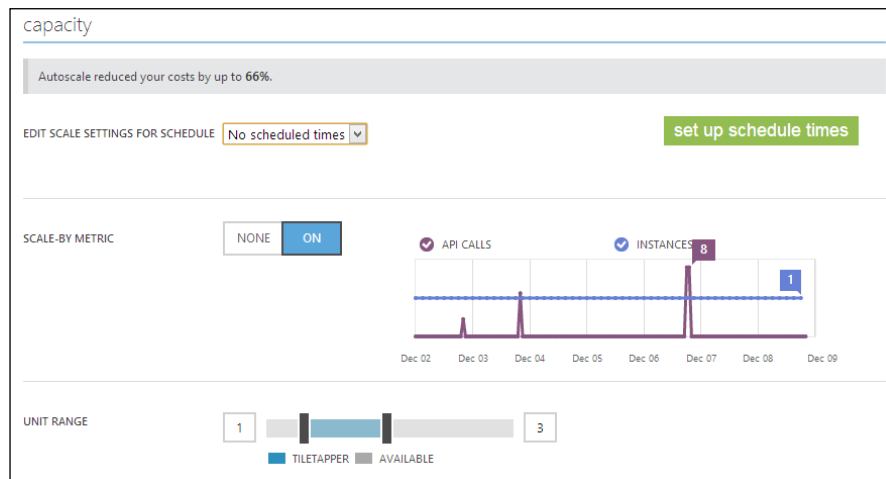
Scale

One of the major features of Windows Azure is scalability. Windows Azure websites, web services, windows services, mobile services, and so on run in virtual machine instances managed by the **Windows Azure Fabric Controller**. This not only provides us with resilience but also allows a service to be scaled across multiple instances to meet the required capacity. We can configure the following features from here:

- **General:** This section allows us to change the **MOBILE SERVICE TIER**, which determines whether certain features can be used. In the **BASIC** and **STANDARD** mode, we can adjust the number of units in operation or auto scaling.
- **Capacity:** If we use a basic or standard service tier, we can configure the number of live units when **SCALE-BY METRIC** is set to **OFF**. These units are always active and will cost a fixed amount all the time.



SCALE-BY METRIC is a feature that allows the number of mobile service instances to increase and decrease automatically to meet the demand on the service. When **SCALE-BY METRIC** is set to **ON**, we can set the upper and lower unit thresholds:



With this configuration, we will incur the highest costs on peak demand when the system scales-up automatically, but it should be more economical than having a fixed number of units always active.

- **SQL Database:** Here, we can change the database capacity if required. Once we move away from the free 20 MB database, we will start incurring costs.

Logs

The logs tab allows us to view logs created by script errors or logging during debugging. We will cover more on this in *Chapter 4, Service Customization with Scripts*.

Summary

So far, we've chosen a subscription, signed up for a Windows Azure account, created our first Windows Azure's Mobile Service, and got a taste of what a mobile service has to offer us. Throughout the book, we'll be looking at these features in a lot more detail and learning how to use them in our applications.

In the next chapter, we're going to start setting up our development environment, get all the tools we will need, look at the portal starter solutions, and hook up an app from scratch.

2

Start Developing with Windows Azure Mobile Services

So far, we've got everything ready in the Windows Azure portal, with an account setup and our first Windows Azure Mobile Service created. Next, we're going to look at the following topics:

- Preparing our development environment
- Creating starter apps from the portal
- Connecting existing apps to our service

Preparing our development environment

Chances are, you're already developing Windows Phone 8 or Windows Store apps so you'll have some of the tools you need, but there are a few extra bits of software you might need for certain features of Windows Azure Mobile Services. If you've not done Windows Phone development before and plan on doing so, definitely read all of this.

Requirement for hardware

For **Windows Store** app development, there is no special hardware requirement. However, to develop apps for **Windows Phone 8**, you need a machine which has specific requirements in order to run the **Hyper-V** phone emulators. The Windows Phone 8 SDK will do a prerequisite check before installation; however, you can read the exact requirements here: [http://msdn.microsoft.com/en-us/library/windowsphone/develop/ff626524\(v=vs.105\).aspx](http://msdn.microsoft.com/en-us/library/windowsphone/develop/ff626524(v=vs.105).aspx).

For phone development, it is always helpful to have a handset to test on. I would advise testing an app on a real device before publishing it, to make sure that everything works. The same goes for Windows 8; although **Surface Pros** and other tablets running full Windows 8 have exactly the same OS as PCs and laptops, it's helpful to test the touch gestures as well as keyboard as Surfaces (formerly called as Surface RTs) run on a different OS designed for ARM devices so that it is useful to have access to a tablet or machine with a touch screen.

Setting up the software

We will mainly use **Visual Studio** for developing Windows Store and Windows Phone 8 applications. Since I started writing this book, Windows 8.1 was made generally available; so, I'll be using Visual Studio Express 2013 for Windows (2012 version was labelled "for Windows 8") and Visual Studio Express 2012 for Windows Phone (when you are reading this, there may be a 2013 version so use that instead). Of course, you can use Professional and Ultimate versions of Visual Studio and you'll need to download SDKs for Windows 8 and Windows Phone 8 project types. All versions of Visual Studio can be downloaded here: <http://www.microsoft.com/visualstudio/eng/downloads>.

When we start looking at scripts, we'll cover how to manage them using Git version control. So, you'll need to install Git for doing this (<http://git-scm.com/downloads>). When I use Git, I prefer to use the GUI; so, if you want to do the same, make sure you select this option when you install. Also, I use the last option in the installer to prevent Git from changing the file line endings for cross-platform projects so that I don't get annoying warnings whenever I check something in.

We will also make use of **NPM** modules in scripts. So, we will need to install `node.js` from here: <http://nodejs.org/>.

Fiddler is a really helpful HTTP debugging tool that we will mention when we look at security. This can be installed from here: <http://fiddler2.com/>.

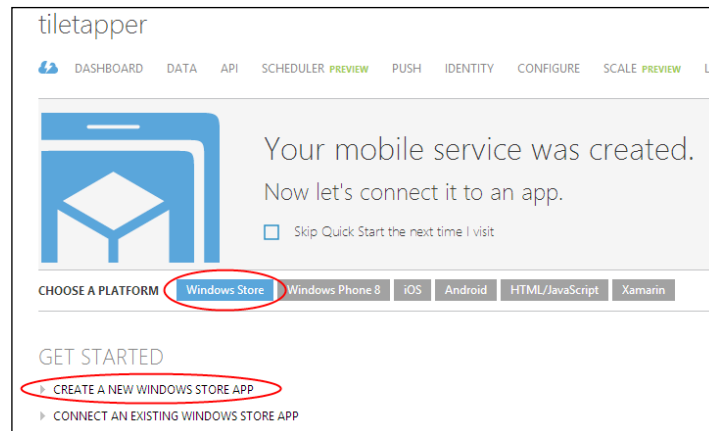
Requirement for store accounts

To publish your apps, you need a store account. You'll also need an account to implement push notifications in Windows Store apps. Unlike Windows Azure Mobile Services, you actually need to pay for these and there is no free option. Previously, you needed separate accounts for Store and Phone apps; however, these have now been merged and only cost 19USD for an individual. You can sign up at: <https://appdev.microsoft.com/StorePortals/en-us/Account/Signup/Start>.

Creating apps from the portal

From the portal, we can download template solutions for Windows Store, Windows Phone 8, iOS, Android, HTML/JavaScript, and Xamarin, which have a working sample of creating a "To-do list" – complete with your app's URL and API key. We're going to take a look at Windows Store app now.

For a Windows Store app, select Windows Store and click on the **CREATE A NEW WINDOWS STORE APP** link:



If you haven't done so already, download Visual Studio. The boilerplate code in the app uses a `ToDoItem` table, so click on the button to create it (you can delete it later if you like). We're going to discuss the C# app, but you can also download a JavaScript app. The downloaded app is in a ZIP folder. Make sure you go to the ZIP file properties and unblock it so we don't have security problems. Unzip the project and open it in Visual Studio. When we examine the solution, we see that it already has the NuGet packages installed for the Windows Azure Mobile Services API.

When we take a look at the `App.xaml.cs` class, we can see that there is a static variable for accessing an instance of `MobileServiceClient` from anywhere in the app. It has the service endpoint and API key configured:

```
namespace TileTapper
{
    /// <summary>
    /// Provides application-specific behavior to supplement the
    default Application class.
    /// </summary>
    sealed partial class App : Application
    {
```



```
// This MobileServiceClient has been configured to communicate
with your Mobile Service's url
// and application key. You're all set to start working with
your Mobile Service!
public static MobileServiceClient MobileService = new
MobileServiceClient(
    "https://tiletapper.azure-mobile.net/",
    "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
);
```

Downloading the example code



You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

MainPage.xaml and MainPage.xaml.cs contain the template code for exercising the `ToDoList` table. Take a look round the code, then run the app, and have a quick play. The app should look something like this:

WINDOWS AZURE MOBILE SERVICES

TileTapper

1

Insert a TodoItem
Enter some text below and click Save to insert a new todo item into your database

2

Query and Update Data
Click refresh below to load the unfinished TodoItems from your database. Use the checkbox to complete and update your TodoItems

☐ Finish chapter 2

☐ Start chapter 3

Once you've inserted a few items, go back to the portal and take a look at the **ToDoList** table in the **DATA** tab:

todoitem

BROWSE SCRIPT COLUMNS PERMISSIONS

id	text	complete	__createdAt	__updatedAt	__version
AD69C27F-74CE-47F6-99...	Finish Chapter 2	true	2013-12-09T20:13:26.714...	2013-12-09T20:13:29.542...	AAAAAAAAAB9Q=
BF3A9BA1-EFCC-43C8-A1...	Start Chapter 3	false	2013-12-09T20:13:38.73+...	2013-12-09T20:13:38.746...	AAAAAAAAAB9Y=

REFRESH TRUNCATE 1 ?

You can use the **TRUNCATE** button to delete all the records you've created.

The Windows Phone 8 app is pretty much identical, so we won't go through it now; but have a look yourself or you may want to look at it instead of the Windows Store version.

The SDK implemented in these template apps exposes the mobile service REST API, which can be consumed by any platform capable of making HTTP requests, and not just ones listed in the portal.

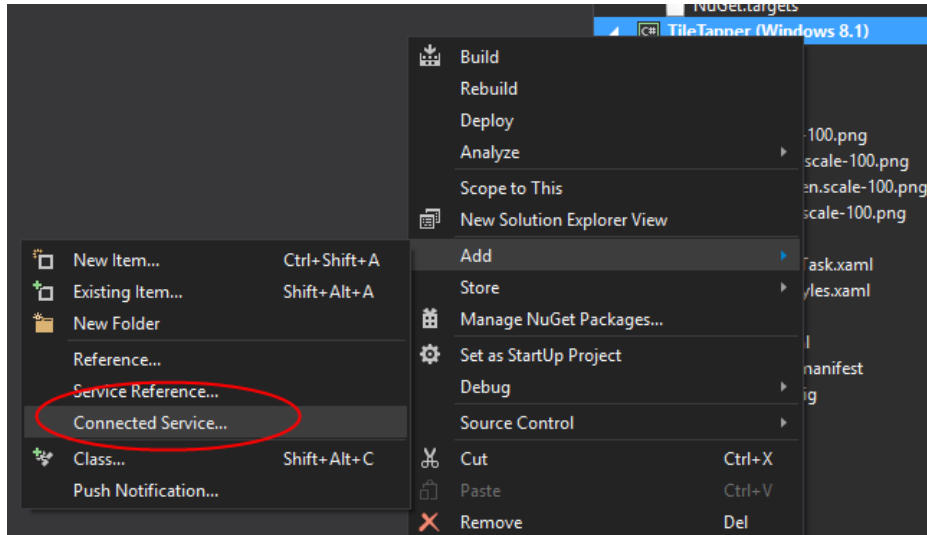
Connecting existing apps to Windows Azure Mobile Services


Connecting existing apps is simple to do. We can add a Connected Service for Visual Studio 2013 solutions and we need to install the Windows Azure Mobile Services SDK NuGet package for Visual Studio 2012, or you can download the source from the following link: <https://github.com/WindowsAzure/azure-mobile-services>. As you can see, the SDK is open source and hosted on GitHub rather than CodePlex which is the usual place Microsoft host SDKs. So, it shows that they're really building a cross-platform service here.

Adding a Connected Service in Visual Studio 2013

To connect to a service, follow these instructions:

1. Right-click a project in the solution explorer, select **Add | Connected Service**:



[ If you have imported a subscription, skip to the last step.]

2. Click on **Import subscriptions** and the **Import Windows Azure Subscriptions** dialog will appear.
3. Click on **Download subscription file**. Your default browser will be launched and the subscriptions file will be downloaded automatically. If you are logged into the portal, this will happen automatically; otherwise, you'll be prompted to log in.
4. Once downloaded, browse to the downloaded file in the **Import Windows Azure Subscriptions** dialog box and click on **Import**.
5. Select the subscription you want to use and click on **OK**.

The SDK NuGet package will be installed into our app and a static `MobileServiceClient` instance will be added to `App.xaml.cs`, in the same way as the app downloaded from the portal.

Manually installing the SDK in Visual Studio 2012 Express for Windows Phone

First, we're going to install the NuGet package into our solution. This can be done from the **NuGet Package Manager** dialog box by right-clicking on the project and selecting **Manage NuGet Packages**; or alternatively, from the **Package Manager Console** by typing the following command:

```
PM> Install-Package WindowsAzure.MobileServices
```

Install the package (accepting the licenses) and we're ready to go.

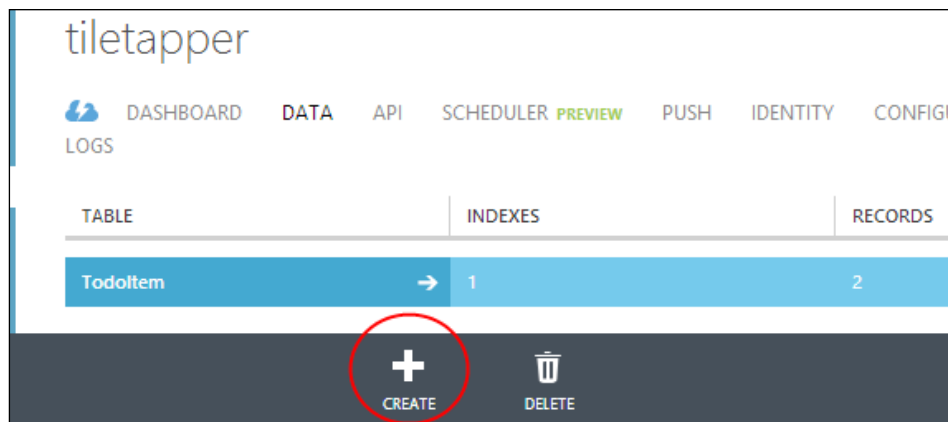


If the install fails, check whether your NuGet Package Manager extension is up-to-date (by going to **Tools | Extensions and Updates | Updates**).

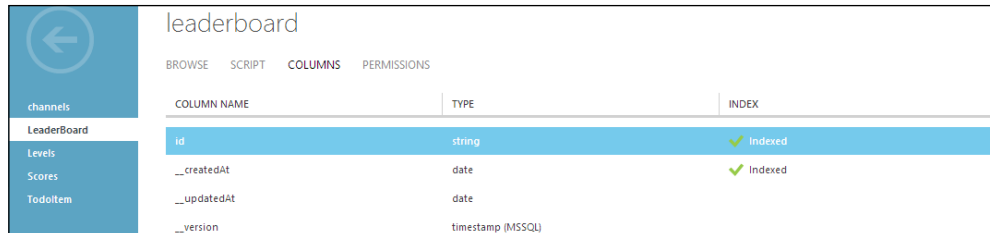
I prefer to implement `MobileServiceClient` in my own `DataService` class. So, I would install the package manually even in Visual Studio 2013 to save cleaning up code in `App.xaml.cs`.

Creating a table

We've got a database, but we need a table to interact with to get started. For the **TileTapper** game, we need a **LeaderBoard** table to keep track of player's high scores. So, we'll create that now. Click on the **CREATE** button on the toolbar in the **DATA** tab:



From the **Create New Table** dialog, enter the table name and for now, leave the default permissions (we'll look at these when we start talking about permissions in the next chapter). By default the database is set to have a dynamic schema. This means that the table adds new columns as it finds them in the inserted data.



leaderboard		
BROWSE SCRIPT COLUMNS PERMISSIONS		
COLUMN NAME	TYPE	INDEX
id	string	✓ Indexed
__createdAt	date	✓ Indexed
__updatedAt	date	
__version	timestamp (MSSQL)	

We can see that we've got a table which already has an indexed **id** column and also **__createdAt**, **__updatedAt**, and **__version** columns for optimistic concurrency.

Writing a model of the table

We'll go back to Visual Studio and write a model for the **LeaderBoard** table that will be used to read and write records to the table. When the database first sees the model, it will create the table columns for us. Here's the code for the model:

```
using System;
using Newtonsoft.Json;
namespace TileTapper.Models
{
    [JsonObject(Title="leaderboard")]
    public class LeaderBoardItem
    {
        [JsonProperty(PropertyName = "id")]
        public string Id { get; set; }

        [JsonProperty(PropertyName = "timeStamp")]
        public DateTime TimeStamp { get; set; }

        [JsonProperty(PropertyName = "name")]
        public string Name { get; set; }

        [JsonProperty(PropertyName = "score")]
        public int Score { get; set; }
    }
}
```

You will notice that there are `JsonObject` and `JsonProperty` attributes on the class and properties. These attributes tell the JSON serializer to use these names instead of the property or class name when the object is serialized, so that we can have different names (I didn't want my item to be called `LeaderBoard` as this didn't make sense), our C# properties in Pascal case (`PascalCase`), and the JSON objects in Camel case (`camelCase`).

Interacting with the table

The next step is to write some code to interact with the `LeaderBoard` service that exposes the table. We're going to start a data service class to contain all the operations we want to perform on the `LeaderBoard` table. I'm steering us towards using an **Model View View-Model (MVVM)** pattern (you can read a bit about MVVM at http://en.wikipedia.org/wiki/Model_View_ViewModel), but we'll try and organise our code so that things are kept simple and our UI code is not littered with data access code. Here's the service with a `GetAll` and `Insert` method:

```
using System;
using System.Threading.Tasks;
using System.Collections.Generic;
using Microsoft.WindowsAzure.MobileServices;
using TileTapper.Models;

namespace TileTapper.Services
{
    public class DataService
    {
        // Our MobileServiceClient instance with Url and
        // application key set
        private static readonly MobileServiceClient _mobileService
            = new MobileServiceClient(
                "https://tiletapper.azure-mobile.net/",
                "0000CZGhLgIKxkrBCFwxSGXKHZPLRq15"
            );

        public async Task<IEnumerable<LeaderBoardItem>> GetAll()
        {
            var table =
                _mobileService.GetTable<LeaderBoardItem>();
            return await table.ToEnumerableAsync();
        }

        public async Task Insert(LeaderBoardItem item)
        {

```

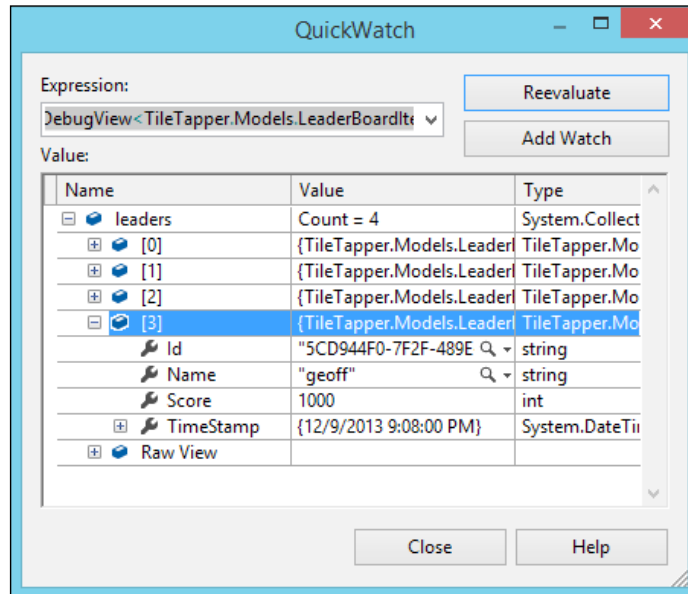
```
        var table =  
            _mobileService.GetTable<LeaderBoardItem>();  
        await table.InsertAsync(item);  
    }  
}  
}
```

The service contains its own instance of the `MobileServiceClient` object that allows us to access the service we created. You can find the URL on the dashboard in the portal and the key under **Manage Keys** on the portal's toolbar. We then interact with the table using the generic `GetTable` method. You'll notice the use of `async` and `Task`, these were introduced in C# 5 and feature heavily in Windows Store and Windows Phone 8 app development. If you're not familiar with asynchronous programming, it's worth having a quick read up on the Web. In C++ and JavaScript, `async` is handled differently.

To give the service a test drive, I created some temporary methods that are called in the app's `MainPage.xaml.cs` constructor to seed the table and examine the contents afterwards:

```
private async void Demo()  
{  
    await this.Seed();  
    await this.GetAll();  
}  
  
private async Task Seed()  
{  
    var service = new DataService();  
    // Seed a few items into the Leader Board  
    await service.Insert(new Models.LeaderBoardItem()  
    {  
        Name = "Tank Man",  
        Score = 885562,  
        TimeStamp = DateTime.Now  
    });  
  
    // A few others removed for brevity  
}  
  
private async Task GetAll()  
{  
    var service = new DataService();  
    var task = await service.GetAll();  
  
    // Materialize leaders so we can have a look  
    var leaders = task.ToList();  
}
```

You'll notice that I'm not setting the `id` field in the models as these will be set by the database. If we put a breakpoint at the bottom of `GetAll`, we can see our four leaders have been created and the IDs are set by the database:



If we now have a look at the table in the portal, we can see the data and the columns that have been created for us:

leaderboard					
BROWSE	SCRIPT	COLUMNS	PERMISSIONS		
id	timeStamp	name	score	__createdAt	__updatedAt
10E4FC3A-B55D-43AA-91...	2013-12-09T21:07:58.415...	Ultimate Fail	0	2013-12-09T21:07:57.772...	2013-12-09T21:07:57.788...
2FD9E522-B276-44F2-980...	2013-12-09T21:07:55.029...	Tank Man	885562	2013-12-09T21:07:56.788...	2013-12-09T21:07:56.788...
36AB6BD0-716B-4871-8F...	2013-12-09T21:07:59.315...	37337	999999999	2013-12-09T21:07:58.694...	2013-12-09T21:07:58.694...
5CD944F0-7F2F-489E-B8B...	2013-12-09T21:08:00.247...	geoff	1000	2013-12-09T21:07:59.71+...	2013-12-09T21:07:59.71+...

Summary

We've now got all the development tools we need installed on our machine, had a look at the starter solutions which can be downloaded from the portal, installed the SDK into our own solution, and started laying down some foundations in code ready for adding some more interesting features.

In the next chapter, we'll look at the security features Windows Azure Mobile Services offers us to protect our data and users.

3

Securing Data and Protecting the User

Security is extremely important for any system that is exposed to the Internet. Using Windows Azure Mobile Services with our applications is no different from any other system; we are exposing our data to the Internet on the server side and our users on the app side.

Windows Azure Mobile Services makes it easy to achieve a secure system by offering us the following features:

- Services are hosted on a highly-secure infrastructure; so we don't have to worry about hardening servers, configuring firewalls, and patching software.
- Granular permission control on individual tables and API methods means that we can tailor permissions down to method level on tables and APIs.
- Authentication is delegated to third-party authentication providers. So we don't need to store user credentials ourselves; we can let someone else take care of this for us.
- With authentication delegation, we don't need to create a full set of user admin UI in our application, which is time-consuming.
- Windows Azure Mobile Services use HTTPS, which means our data is encrypted between the client and server.

In this chapter, we'll discuss permissions, setting up an authentication provider, writing code to authenticate our users, and accessing the REST API with the master key.

Configuring permissions

By default, table and API methods allow requests that are made with the app key. This behavior can be changed to one of the following options per method:

- **Everyone:** This is the least secure option for a method as it allows anybody who knows your service URL to call it
- **Anybody with the Application Key:** This is the default option
- **Only Authenticated Users:** If this option is chosen, requests must be authenticated with one of the providers configured in the **IDENTITY** tab
- **Only Scripts and Admins:** If this option is chosen, only requests authenticated with the master key or from internal scripts will be allowed

Rules for choosing permissions

The following is a list of rules to help choosing permissions:

- If a user doesn't need to use a service method and we only need to perform administrative tasks, apply **Only Scripts and Admins**
- If a user requires the `Insert`, `Update`, and `Delete` methods, apply **Only Authenticated Users** and make sure only the user's data is available to them in the `Read` methods with custom scripts
- `Read` on public tables can have **Anybody with the Application Key** if we aren't tracking the user
- Don't use the **Everyone** option unless you want anyone with the service URL to use your service method

The `leaderBoard` table has **INSERT PERMISSION** set to **Only Authenticated Users** and **READ PERMISSION** set to **Anybody with the Application Key** because all users have access to this table and we're not tracking their credentials. **UPDATE PERMISSION** and **DELETE PERMISSION** is set to **Only Scripts and Admins** because they're not used, and we don't want these methods being used by anybody except the administrators, as shown in the following screenshot:

leaderboard

BROWSE SCRIPT COLUMNS **PERMISSIONS**

table permissions

INSERT PERMISSION	Only Authenticated Users
UPDATE PERMISSION	Only Scripts and Admins
DELETE PERMISSION	Only Scripts and Admins
READ PERMISSION	Anybody with the Application Key

Authentication providers

Windows Azure Mobile Services support the following OAuth2 authentication providers:

- **Microsoft:** <http://msdn.microsoft.com/en-us/live//default.aspx>
- **Twitter:** <https://dev.twitter.com/apps>
- **Facebook:** <https://developers.facebook.com/apps>
- **Google:** <https://code.google.com/apis/console>

Any of the preceding can be implemented by creating an application with the provider that will provide you with a client ID and secret key. These details are then entered in the **Identity** tab of the portal.

Authentication

We should not think that if a user has been authenticated by a provider, they can be trusted with all our services; they can't. Authentication just means that the users are who they say they are, and we can use their identity to manage their data. We should only allow them to access services they need, and only allow them to read, update, and delete their own data and read public data.

Registering for Windows Live Connect Single Sign-on

Go to the **Live Connect Developer Center** at <http://msdn.microsoft.com/en-us/live//default.aspx>, click on **My apps**, and enter the app's details, as shown in the following screenshot:

Live Connect Developer Center Geoff Webber-Cross | Sign out

Home My apps Docs Interactive SDK Downloads Support Showcase

My applications

Connect your application to Windows Live

Provide the name of your application that users will see.

Application name*

Language*

Use letters, digits and underscores only. 129-character limit.

Select your application's primary language.

Clicking **I accept** means you agree to the Live Connect [terms of use](#). Read [Privacy & Cookies](#).

I accept **Cancel**

Once you have accepted the terms and conditions, you need to configure the application details, as shown in the following screenshot:

TileTapper

Settings

- Basic Information
- API Settings**
- Localisation

Your application was created successfully, but it must be configured before you can use it.
If your app is a mobile or desktop client application, click "Yes" below.
If your app is a web application, enter its Redirect domain.

Client ID: 00000000
This is a unique identifier for your application.

Client secret (v1): h-Ac...
For security purposes, don't share your client secret with anyone.
[Create a new client secret](#)

Redirect domain:
Live Connect enforces this domain in your OAuth 2.0 redirect URI that exchanges tokens, data and messages with your application. You only need to enter the domain, for example <http://www.contoso.com>

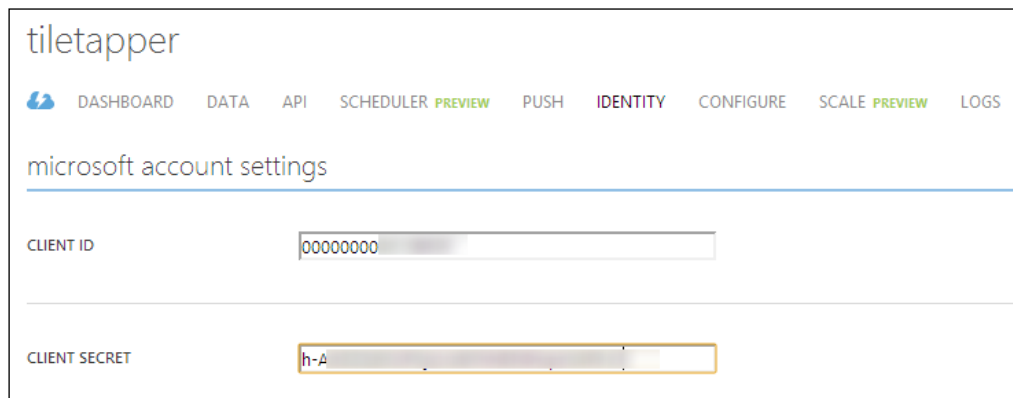
Mobile client app:
☒ Yes ☐ No
Mobile client applications use a different OAuth 2.0 authentication flow. Only select "Yes" if your app is a mobile app. [Learn More](#)

Restrict JWT Issuing
☐ Yes ☒ No
Limits the issuing of JWT tickets for your domain to exclusively your Windows 8 application.

Save **Cancel**

Enter your Windows Azure Mobile Service's URL in the **Redirect domain** field and select **Yes** for **Mobile client app**. I selected **No** for **Restrict JWT Issuing** because I want my Windows 8 and Windows Phone 8 app to use the same authentication provider application. Restricting **JSON Web Token (JWT)** is a security mechanism for allowing just one application to use the Live Connect application.

Once this is done, copy **Client ID** and **Client secret (v1)** to the **microsoft account settings** section on the **IDENTITY** tab in the Windows Azure Mobile Services portal and click on **SAVE**, as shown in the following screenshot:



Authentication in the app

If you've created apps before that need to authenticate a user with OAuth2 in order to use services from providers such as Twitter and Facebook, you'll know that it's not straightforward. It has steps such as launching a web page for the user to log in and collecting credentials from browser redirects. Authentication with Windows Azure Mobile Services couldn't be simpler. The `MobileServiceClient` class has a `LoginAsync` method that does everything for us.

Logging in

To log in, we use the `MobileServiceClient` class instantiated with our app key and service URL that we saw in the previous chapter. We simply call the `LoginAsync` method with the auth provider type we want to use. It will log us in and return a `MobileServiceUser` object that contains a user ID and auth token, as shown in the following code snippet:

```
// Login
var user = await this._mobileService
    .LoginAsync(MobileServiceAuthenticationProvider.MicrosoftAccount);
```

In a Windows 8 app, the `LoginAsync` method launches a login page that contains the provider's login web page for the user to enter their details, as shown in the following screenshot:

Microsoft account

Sign in

Microsoft account [What's this?](#)

Password

☐ Keep me signed in

[Can't access your account?](#)

Don't have a Microsoft account? [Sign up](#)

When we examine the user object that is returned, this is what we see:

Name	Value	Type
user	{Microsoft.WindowsAzu	Microsof
MobileServiceAuthenticationToken	"eyJhbGciOiJIUzI1N	string
UserId	"MicrosoftAccount	string

Once we've done this, the `MobileServiceClient` object has the `CurrentUser` property set to this user, and the details will be used to authenticate our requests, as shown in the following screenshot:

Name	Value	Type
_mobileService	{Microsoft.WindowsAzure.Mobil	Microsof
ApplicationKey	"IWwHCZGhLgIKxkrBCFwx	string
ApplicationUri	{https://tiletapper.azure-mobile	System.l
CurrentUser	{Microsoft.WindowsAzure.Mobil	Microsof
MobileServiceAuthenticationToken	"eyJhbGciOiJIUzI1NiIsInR5c	string
UserId	"MicrosoftAccount:93c4f80	string

If the authentication fails, an `InvalidOperationException` will be thrown with the `Error: Unauthorized` message so that we can catch it, as shown in the following code snippet:

```

catch (InvalidOperationException ioex)
{
    // Task has failed because it was unauthorized try again
    if (ioex.Message == "Error: Unauthorized")
    {

    }
}

```

Storing credentials

This is all very good, but we don't want our users to log in every time they open the app, or when the app resumes after suspension—it would not make for a good user experience! We can get around this by storing the user credentials when they log in, and then retrieving them and applying them to `MobileServiceClient` whenever required. This is achieved by the following method:

```

public static async Task<bool> Login()
{
    // First have a look and see if we have the user's token
    var userId = StorageHelper.GetSetting<string>(USER_ID, null);
    var userToken = StorageHelper.GetSetting<string>(USER_TOKEN,
        null);

    bool success = true;

    if (userId != null && userToken != null)
    {
        // Create user and apply to client
        var user = new MobileServiceUser(userId);
        user.MobileServiceAuthenticationToken = userToken;
        _mobileService.CurrentUser = user;
    }
    else
    {
        try
        {
            // Login
            var user = await _mobileService.LoginAsync(_provider);

            // Store credentials
            StorageHelper.StoreSetting(USER_ID, user.UserId, true);
            StorageHelper.StoreSetting(USER_TOKEN,
                user.MobileServiceAuthenticationToken, true);
        }
        catch (InvalidOperationException)
        {
            // Something has gone wrong, most likely user cancelled by

```



```
        // backing-out
        success = false;
    }
}

return success;
}
```

StorageHelper is a helper class I wrote to read and write typed settings to storage. You can get it in the code samples.

Now, our users don't have to log in every time they run the app. But what will happen when the token expires? Some OAuth2 providers actually tell us the expiry date of the token, but it's not available to us in MobileServiceUser object. What we can do is look for a request that is failing because it is unauthorized or has expired, and then ask the user to log in. I put together this helper method which takes Task wrapped in a Func so that the task can be executed again if it fails, as shown in the following code snippet:

```
protected async Task<T> ExecuteAuthenticated<T>(Func<Task<T>> t, int
    retries = 1)
{
    int retry = 0;
    T retVal = default(T);

    while (retry <= retries)
    {
        // If we have no current user, login
        if (_mobileService.CurrentUser == null)
        {
            // If login fails return default
            if (!await Login())
                return retVal;
        }

        // Try and execute task
        try
        {
            retVal = await t();
            break;
        }
        catch (InvalidOperationException ioex)
        {
            // If task has failed because it was unauthorised try again
            if (ioex.Message == "Error: Unauthorized" || ioex.Message ==
                "Error: The authentication token has expired.")
```

```
        {  
            Logout();  
        }  
  
        retry++;  
    }  
}  
  
return retVal;  
}
```

We can now make any request authenticated, as shown in the following code snippet:

```
public async Task<IEnumerable<LeaderBoardItem>> GetAll()  
{  
    // Make sure we're authenticated by passing the task into  
    // ExecuteAuthenticated  
    return await this.ExecuteAuthenticated(async () =>  
    {  
        var table = _mobileService.GetTable<LeaderBoardItem>();  
        return await table.ToEnumerableAsync();  
    });  
}
```

Notice, we've modified the `GetAll` method in the `LeaderBoard` service and have not changed its signature. So, we haven't touched our UI code, and we have now automatically authenticated all our requests. Pretty cool!

Logging out

The `MobileServiceClient` class has a `Logout` method that doesn't seem to do anything other than clear the `CurrentUser` property. It doesn't void the token with the provider when it is called. If we're storing the user token, we'll also need to clear these too so that the app doesn't log the user back in when it relaunches. The `Logout` method does this for us, as shown in the following code snippet:

```
public void Logout()  
{  
    this._mobileService.Logout();  
  
    // Clear credentials  
    StorageHelper.StoreSetting(USER_ID, null, true);  
    StorageHelper.StoreSetting(USER_TOKEN, null, true);  
}
```

The DataServiceBase class

Now that we've got all the things we need to log the user in and out, it would be nice to wrap it all up so that it's common for all the data services we want to create. To do this, I've created a base class which has a static instance of `MobileServiceClient` and the methods we've just discussed, as shown in the following code snippet:

```
using System;
using System.Threading.Tasks;
using System.Collections.Generic;
using Microsoft.WindowsAzure.MobileServices;
using TileTapper.Models;
using TileTapper.Helpers;
namespace TileTapper.DataServices
{
    public abstract class DataServiceBase
    {
        private const string USER_ID = "USER_ID";
        private const string USER_TOKEN = "USER_TOKEN";

        // Our MobileServiceClient instance with Url and application
        // key set
        protected readonly static MobileServiceClient _mobileService =
            new MobileServiceClient(
                "https://tiletapper.azure-mobile.net/",
                "IWwHCZGhLgIKxkrBCFwxSGXKHZPLRq15"
            );

        protected static MobileServiceAuthenticationProvider _provider
            = MobileServiceAuthenticationProvider.MicrosoftAccount;

        protected async Task<T> ExecuteAuthenticated<T>(
            Func<Task<T>> t, int retries = 1)
        {
            int retry = 0;
            T retVal = default(T);

            while (retry < retries)
            {
                // If we have no current user, login
                if (_mobileService.CurrentUser == null)
                {
                    // If login fails return default
                    if (!await Login())
                        return retVal;
                }
            }
        }
    }
}
```

```
// Try and execute task
try
{
    retVal = await t();
    break;
}
catch (InvalidOperationException ioex)
{
    // If task has failed because it was unauthorised try
    // again
    if (ioex.Message == "Error: Unauthorized" ||
        ioex.Message == "Error: The authentication token has
        expired.")
    {
        Logout();
    }
    retry++;
}
}

return retVal;
}

public static async Task<bool> Login()
{
    // First have a look and see if we have the user's token
    var userId = StorageHelper.GetSetting<string>(USER_ID,
        null);
    var userToken = StorageHelper.GetSetting<string>(USER_TOKEN,
        null);

    MobileServiceUser user = null;

    if (userId != null && userToken != null) {

        // Create user and apply to client
        user = new MobileServiceUser(userId);
        user.MobileServiceAuthenticationToken = userToken;
    }
    else
    {
        try
        {
            // Login
            user = await _mobileService.LoginAsync(_provider);

            // Store credentials
            StorageHelper.StoreSetting(USER_ID, user.UserId, true);
        }
        catch { }
    }
}
```

```
        StorageHelper.StoreSetting(USER_TOKEN,
            user.MobileServiceAuthenticationToken, true);
    }
    catch (InvalidOperationException)
    {
        // Something has gone wrong, most likely user cancelled
        // by backing-out
    }
}

if (user != null)
{
    _mobileService.CurrentUser = user;
    return true;
}

return false;
}

public static void Logout()
{
    _mobileService.Logout();

    // Clear credentials
    StorageHelper.StoreSetting(USER_ID, null, true);
    StorageHelper.StoreSetting(USER_TOKEN, null, true);
}
}
}
```

Now, our data services can inherit from this base class, which means that they are really neat and only concerned with data operations – not security. This is shown in the following code snippet:

```
using System;
using System.Threading.Tasks;
using System.Collections.Generic;
using Microsoft.WindowsAzure.MobileServices;
using TileTapper.Models;
using TileTapper.Helpers;

namespace TileTapper.DataServices
{
    public class LeaderBoardService : DataServiceBase
    {
        /// <summary>
        /// Gets all LeaderBoardItems
        /// </summary>
```

```

    /// <returns>Task to get an enumerable collection of
    /// LeaderBoardItem</returns>
    public async Task<IEnumerable<LeaderBoardItem>> GetAll()
    {
        // Make sure we're authenticated by passing the task into
        // ExecuteAuthenticated
        return await this.ExecuteAuthenticated(async () =>
        {
            var table = _mobileService.GetTable<LeaderBoardItem>();
            return await table.ToEnumerableAsync();
        });
    }
}

```

REST API and the master key

So far, we've seen the app key in action in our app, but we have not really said much about the master key. The master key allows us to access tables and APIs with authentication protection, without authenticating it against our authentication provider. Because the master key has this capability, it must not be distributed with the mobile applications.

It is handy for administrative tasks as we don't need to implement OAuth2 workflow to access the services. Also, there is a useful feature that allows us to suppress custom scripts implemented on table methods, so we can get base-level CRUD operations on the table without any user customizations such as filtering by user or validation affecting the results.

HTTP requests are authenticated with the following optional headers:

- X-ZUMO-APPLICATION: Application key
- X-ZUMO-AUTH: User auth token
- X-ZUMO-MASTER: Master key

In this example, I used Fiddler (<http://fiddler2.com/>) to compose some HTTP requests (you can use any HTTP debugging tool you like). We'll preform a GET request on an authentication-protected table (I temporarily changed the LeaderBoard table for this example).

If we just use our app key, as shown in the following request:

```

GET https://tiletapper.azure-mobile.net/tables/
leaderboard HTTP/1.1
X-ZUMO-APPLICATION: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Host: tiletapper.azure-mobile.net

```

We get a 401 Unauthorized response, as shown in the following:

```
HTTP/1.1 401 Unauthorized
Cache-Control: no-cache
Content-Length: 42
Content-Type: application/json
Server: Microsoft-IIS/8.0
x-zumo-version: Zumo.Main.0.1.6.4247.Runtime
X-Powered-By: ASP.NET
Set-Cookie: ARRAffinity=3b009d5d3272fba37561fb551f1b8cf912175fe784c5b1c8ca93e16259dc3f19;Path=/;Domain=tiletapper.azure-mobile.net
Set-Cookie: WAWebSiteSID=b86a3feb64a441dbbfaa4b72a1704ea; Path=/;
HttpOnly
Date: Tue, 10 Dec 2013 10:31:36 GMT

{"code":401,"error":"Error: Unauthorized"}
```

Then, if we use our master key as shown in the following request:

```
GET https://tiletapper.azure-mobile.net/tables/
  leaderboard HTTP/1.1
X-ZUMO-MASTER: YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY
Host: tiletapper.azure-mobile.net
```

We get a 200 OK response and our JSON data, as shown in the following:

```
Response:
HTTP/1.1 200 OK
Cache-Control: no-cache
Content-Length: 468
Content-Type: application/json
Server: Microsoft-IIS/8.0
x-zumo-version: Zumo.Main.0.1.6.4247.Runtime
X-Powered-By: ASP.NET
Set-Cookie: ARRAffinity=3b009d5d3272fba37561fb551f1b8cf912175fe784c5b1c8ca93e16259dc3f19;Path=/;Domain=tiletapper.azure-mobile.net
Set-Cookie: WAWebSiteSID=9d23b57f4f0447b080d0eb78ed69b328; Path=/;
HttpOnly
Date: Tue, 10 Dec 2013 10:38:26 GMT

[{"id":"2FD9E522-B276-44F2-9801-A0007B1E1286","timeStamp":"2013-12-09T21:07:55.029Z","name":"Tank Man","score":885562},{ "id":"10E4FC3A-B55D-43AA-9104-850A99868C3F","timeStamp":"2013-12-09T21:07:58.415Z","name":"Ultimate Fail","score":0},{ "id":"36AB6BD0-716B-4871-8FD8-04B9E43A7DB7","timeStamp":"2013-12-09T21:07:59.315Z","name":"37337","score":999999999},{ "id":"5CD944F0-7F2F-489E-B8BC-512FDDB764E6","timeStamp":"2013-12-09T21:08:00.247Z","name":"geoff","score":1000}]
```

To suppress scripts and go straight into the table, we add the `noscript` parameter as shown in the following URL:

```
https://tiletapper.azure-mobile.net/tables/leaderboard?noscript=true
```

The API supports OData queries, so we can build pretty flexible admin applications. The table methods have the following HTTP methods:

- **Query:** GET
- **Insert:** POST
- **Update:** PATCH
- **Delete:** DELETE

The `POST` and `PATCH` methods place JSON in the request body. In code, this is achieved by writing JSON text into the request stream before reading the response stream.

Summary

We've talked about the importance of security, discussed the options available to control access to our services, and implemented authentication in our app using Windows Live Connect. We've also implemented a base class for managing login, logout, and storing user credentials so the users don't have to log in repeatedly.

In the next chapter, we are going to learn how to customize our service behavior by using scripts.

4

Service Customization with Scripts

When we create a table, we get a set of methods (*Query*, *Insert*, *Update*, and *Delete*). For many implementations, these methods will be fine as they are, but we can also change their behavior with JavaScript scripts. When we first create a table, we get a set of stubbed scripts, which we can modify to do things such as validate and manipulate our data and filter data for authenticated users.

Just like modifying table methods, we can also create our own API methods outside the scope of a table's operations to do anything we like.

Scripts can access tables, trigger push notifications, make HTTP requests, or do anything we like with third-party libraries using NPM modules. The portal has a fantastic script editor with *intelliSense*, but we can also pull a copy of all the scripts to work on locally using Git version control.

In this chapter, we'll look at some examples which try and incorporate a number of features in one go rather than examining things individually out of context.

Understanding table scripts

Each table's *Insert*, *Update*, *Delete*, and *Read* methods can be modified. All the methods take a *user* and *request* parameter. The method signatures look like the following:

```
function del(id, user, request)
function insert(item, user, request)
function read(query, user, request)
function update(item, user, request)
```

The following parameters are used in the method signatures:

- `id`: This is the ID of an item to be deleted with the `delete` method.
- `item`: This is the item object to be inserted or updated. It will have the same properties as the model that we created in the app.
- `query`: This is the OData query expression for reading data with a query.
- `user`: This is the user object with user ID, level (admin, anonymous, and authenticated), and access tokens properties.
- `request`: This contains execute methods that execute the default action for the method and a `respond` method that returns the response.

Level-insert table script example

I've created a table to store game levels called `Levels`. When the app starts, it loads the levels from the service. This script is implemented in the `insert` method of the `Levels` table to validate the data and check if the board cells are square. The steps are as follows:

1. First we do a null check:

```
if(item === null || item.name === null || item.time === null){
    valid = false;
}
```

2. Next check if the `cells` property is square (and not null):

```
if(valid && item.cells !== null){
    var sqrt = Math.sqrt(item.cells.length);
    if(Math.pow(sqrt, 2) !== item.cells.length){
        valid = false;
    }
}
else {
    valid = false;
}
```

3. Finally, we execute if valid or respond with a bad request:

```
if(valid){
    request.execute({
        success: function(results) {
            console.log("Inserted level");
        },
        error: function(error) {
            console.error(error);
        }
    });
}
```

```

    }
  });
} else {
  request.respond(statusCodes.BAD_REQUEST);
}

```



You can see the full script in the samples.

Score-insert script example

The scores table holds user scores. Therefore, we need to make sure we track the authenticated user's ID so that we can filter their results for other operations.

The insert script populates the owner property from `user.userId`:

```

function insert(item, user, request) {

  item.owner = user.userId;
  request.execute();
}

```

Score-read script example

Because we only want to return results for the calling user, we can filter just their data as follows:

```

function read(query, user, request) {

  query.where({
    owner: user.userId
  });
  request.execute();
}

```

For both the `Score` methods, users must be authenticated. For the `TileTapper` game, the user doesn't need to use the `update` or `delete` method, so I've set these to be **Only Authenticated Users**. But, if you need to use them, the item's owner should be checked before execution.

API scripts

API scripts can be used to do things outside the scope of a table's method, such as making HTTP requests to call other web services and performing push notification requests (this will be discussed in the next chapter). We can still access tables through the `request.service.tables` object, which exposes our tables and all their methods. However, these methods go directly into the table and not through the API. Hence, any table script modifications will be bypassed.

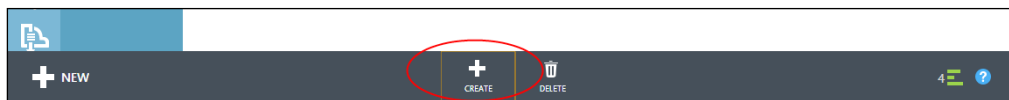
API scripts support the following five HTTP methods:

- GET
- POST
- PUT
- PATCH
- DELETE

Creating an API script

To create an API script, follow this procedure:

1. Go to the API tab in the portal and click on the **CREATE** button:



2. Enter the name and choose the permissions:

A screenshot of a dialog box titled 'MOBILE SERVICES: API' with the subtitle 'Create a new custom API'. It contains a text input field for 'API NAME' with the value 'HighScore'. Below this, a message states: 'You can set a permission level against each HTTP method for your custom API.' There are six dropdown menus for permissions: 'GET PERMISSION', 'POST PERMISSION', 'PUT PERMISSION', 'PATCH PERMISSION', and 'DELETE PERMISSION'. Each dropdown is currently set to 'Anybody with the Application Key'. A checkmark icon is in the bottom right corner.

3. Once created, select it from the API list in the portal. A stubbed GET and POST method is created for you:

```
exports.post = function(request, response) {
  // Use "request.service" to access features of your mobile
  service, e.g.:
  //   var tables = request.service.tables;
  //   var push = request.service.push;

  response.send(statusCodes.OK, { message : 'Hello World!' });
};

exports.get = function(request, response) {
  response.send(statusCodes.OK, { message : 'Hello World!' });
};
```

4. You can modify or delete these methods. To add different methods, just add a new exports method with the HTTP method you require.

High-score API script

The TileTapper game uses an API called HighScore. The high score POST script adds a new item to the leaderboard table, validates that it is the highest score, and calls an external web service:

1. First, we're going to grab the JSON object from the request body:

```
exports.post = function(request, response) {

  // Get item from request body
  var item = request.body;
```

2. Next, we query the leaderboard table to get the highest score by performing an `orderByDescending` and `take(1)` operation and then calling `read`, which takes a success and error function (I've chopped the inside out of the success function so you can see it in one.). The error function logs the error and returns an error response (400), shown as follows:

```
// Get high score
var leaderboardTable = request.service.tables.
getTable('leaderboard');
leaderboardTable
.orderByDescending('score').take(1)
.read({
  // Read success function
  success: function(results) {
```

```
        // Success code removed for brevity
    },
    // Read error function
    error: function(err) {
        console.error(err);
        response.send(400);
    }
  });
```

3. When the read is successful, we go on to get the value from the results, then check if the user score is actually higher. Again, if it is not, we log an error and return an error response (400), shown as follows:

```
var highScore = 0;

// Try and get high score
if(results.length > 0){
    highScore = results[0].score;
}

// If new score is higher execute
if(item.score > highScore){

    // Success code removed for brevity
}
else { // Otherwise return failure
    var msg = "Score " + item.score + ", is lower than high
score " + highScore;
    console.error(msg);
    response.send(statusCodes.BAD_REQUEST, msg);
}
```

4. If the score is higher, we insert the item into the table and use success and error functions again:

```
    // Insert into table
    leaderboardTable.insert(item, {success:
function(results)
{
    // Success code removed for brevity
},
    error: function(err) {
        console.error(err);
        response.send(statusCodes.BAD_REQUEST, err);
    }});
```

5. Once inserted, we're going to send the new high score to an external web service. I published an **MVC Web API** service to an Azure website to test it. We load an NPM package (already installed) called `request`, which simplifies HTTP operations in JavaScript, by using the `require` method:

```
// Send score to external web service
var httpRequest = require("request");
var url = "http://tiletapperadmin.azurewebsites.net/api/
leaderboard";
httpRequest.post({
  url: url,
  json: item
}, function(err, response, body) {
  if (err) {
    console.error("Error connecting to admin service");
  } else if (response.statusCode !== 200) {
    console.error("Error posting to admin service");
  } else {
    console.log("Posted to admin service, response: " +
JSON.stringify(body));
  }
});

response.send(statusCodes.OK, results);
```

Finally, we've got to the bit of the script where we can send an OK result (200) along with the inserted item and its ID set. Note that I've not worried about the output of the result of the web request. The results are logged, but if it fails, I don't want to return an error as we've still successfully inserted the item in the table.

API methods can be called using the `InvokeApiAsync` method, which has a number of overloads for whether you want to pass in an object, return an object, or pass in queries. We can call this method using the following code:

```
var result = await _mobileService.InvokeApiAsync<HighScore,
LeaderBoardItem>("highScore", item);
```

Here, `item` is an instance of the `highScore` model:

```
[JsonObject(Title = "highScore")]
public class HighScore
{
  [JsonProperty(PropertyName = "name")]
  public string Name { get; set; }

  [JsonProperty(PropertyName = "score")]
  public int Score { get; set; }
}
```


Script debugging and logs

Scripts can easily be debugged using the `console` object, which has the following methods:

- `console.log(formatString, obj1, obj2, ...)`: This method logs at info level
- `console.info(formatString, obj1, obj2, ...)`: This method logs at info level
- `console.warn(formatString, obj1, obj2, ...)`: This method logs at warn level
- `console.error(formatString, obj1, obj2, ...)`: This method logs at error level

These methods output a single log entry that can be viewed under the **LOGS** tab in the portal. All the methods can be called with a single string or a formatter and object arguments, shown as follows:

- **Number (%d)**: `console.log("Board size: %d", size);`
- **String (%s)**: `console.log("Board: %s", board);`
- **JSON (%j)**: `console.log("Level JSON: %j", level);`

While working with scripts, we soon learn that logging is our friend. Every time we make a mistake (which we will as we can't debug them in our own IDE), errors will be logged, which we can view under the **LOGS** tab in the portal. Logged errors are generally pretty helpful, telling you which script failed and what the error was.

I had the following error on the `api/highscore.js` script:

```
{ _super: undefined, message: 'A value cannot be specified for  
property \'id\'', code: 'BadInput' }
```

It was telling me that I was trying to insert an object with the `id` property set into the leaderboard table.

Scheduling

From the **SCHEDULER** tab in the portal, it's possible to write scripts to be run on a schedule (or on demand) to perform tasks such as cleaning up data or sending push notifications.



Note that free and basic subscriptions are allowed one task and standard subscriptions are allowed 10.

For the TileTapper game, I created a scheduled script to create daily game levels:

```
function DailyLevel() {
    // Set board size
    var min = 3;
    var max = 10;

    var size = Math.floor((Math.random()*(max-min))+min);

    // Create board
    var board = "";
    for(var i = 0; i < size; i++)
    {
        for(var j = 0; j < size; j++)
        {
            if(Math.random() < 0.5){
                // Active tile
                board += "1";
            }
            else{
                // Inactive tile
                board += "0";
            }
        }
    }

    // Set allowed time
    min = 100;
    max = 3000;

    var time = Math.floor((Math.random()*(max-min))+min);

    // Get reference to Levels table
    var levelsTable = tables.getTable('levels');

    // Add level
    levelsTable.insert({
        name: "Level X",
        cells: board,
        time: time
    });
}
```

The script makes use of the standard JavaScript `Math` object to randomize board size and time allowance, and then uses the API-specific `tables` object to insert the level into the table.

Working locally with Git

We can work on scripts in the portal if we like; however, we can also pull a copy if we want to work locally or for a backup using Git version control. I personally use the Git GUI; however, I don't want to waste pages with screenshots of this, so we'll talk about using Git Bash (the console)!

Pulling the repository

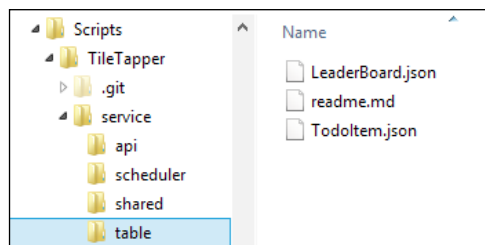
First, we need to get a copy of the repository onto our local machine. To do this, follow this procedure:

1. We need to set up the service's repository Git credentials. On the dashboard, click on the **Set up source control** button and enter some credentials for Git authentication.
2. Create a folder somewhere for the repository (I put mine in one of the Visual Studio projects, so I can work on the scripts easily in Visual Studio). Next, launch **Git Bash** by right-clicking on the folder and selecting **Git Bash** or launching Bash and setting the path to the directory you want.
3. Type the following command into Bash (You can copy the URL from the **GIT URL** setting under the **CONFIGURE** tab):

```
$ git clone https://your_service.scm.azure-mobile.net/Your_Service.git
```

Enter the user name and password when prompted.

4. We should now have a full copy of the service's scripts in our directory:



5. If you've pulled them under one of your projects, you can add the directory into your solution and even check them in to TFS if you're using it.

Updating our repository

When we add or change tables or other scripted items in our service through the portal, we can call a `pull` to update our local repository:

```
$ git pull origin master
```

Enter the username and password when prompted. If there are any conflicts, edit the conflicting files and call `commit`.

Adding scripts manually

I manually added a script named `LeaderBoard.insert.js` to modify the insert behavior of the `LeaderBoard` table:

```
function insert(item, user, request) {  
    request.execute();  
    console.log(item);  
}
```

This will asynchronously insert the item into the table and log the JSON item object to a log file, which we can view in the portal.

We need to add this to the repository by calling an `add` to add the file to the repository:

```
$ git add service/table/LeaderBoard.insert.js
```

Or we can use the following:

```
$ git add *
```

Once added, we can commit the change and add a comment:

```
$ git commit -m "Added LeaderBoard insert script"
```

Pushing back changes

Once we've done some work and committed everything, we can go and push the changes back to the service by calling a `push`:

```
$ git push origin master
```

Enter the username and password when prompted. We can now see that any changes made are reflected in the portal.

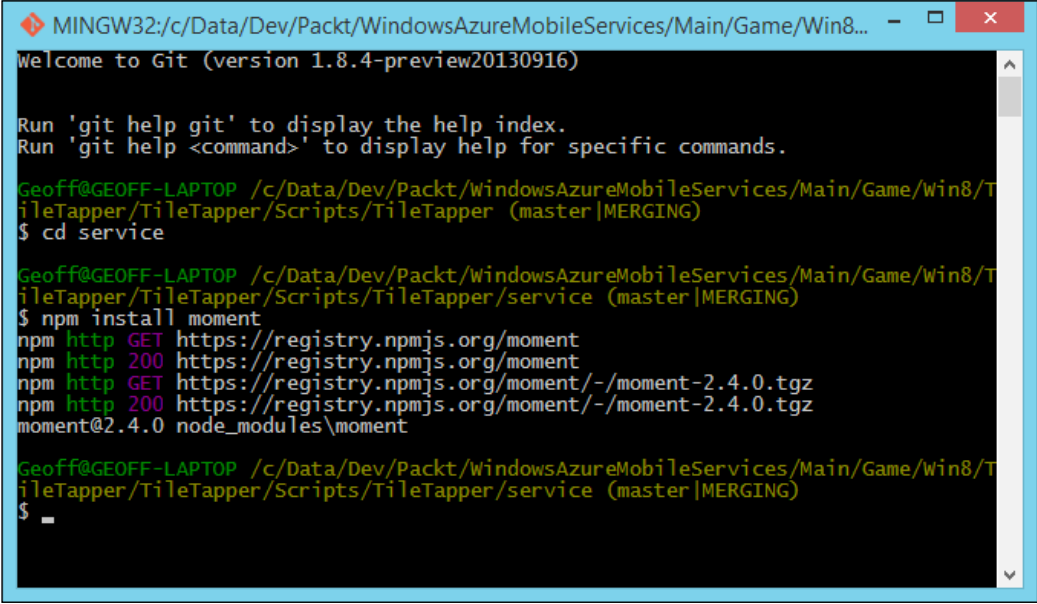
Implementing NPM modules

It's possible to make use of existing script libraries that have a **Node NPM module**. For the TileTapper game, I decided to use the `moment.js` library to easily get a formatted date string for the level name. To use a NPM module in your scripts, follow the following procedure:

1. Make sure you have installed `node.js` (see *Chapter 2, Start Developing with Windows Azure Mobile Services*).
2. Launch Git Bash and navigate to your repository.
3. Update your repository (commit any changes first):

```
$ git pull origin master
```
4. Navigate to the `service` directory.
5. Install the NPM package with the following command:

```
$ npm install package_name
```
6. We should see the following results:



```
MINGW32:/c:/Data/Dev/Packt/WindowsAzureMobileServices/Main/Game/Win8... - [X]
Welcome to Git (version 1.8.4-preview20130916)

Run 'git help git' to display the help index.
Run 'git help <command>' to display help for specific commands.

Geoff@GEOFF-LAPTOP /c:/Data/Dev/Packt/WindowsAzureMobileServices/Main/Game/Win8/T
ileTapper/TileTapper/Scripts/TileTapper (master|MERGING)
$ cd service

Geoff@GEOFF-LAPTOP /c:/Data/Dev/Packt/WindowsAzureMobileServices/Main/Game/Win8/T
ileTapper/TileTapper/Scripts/TileTapper/service (master|MERGING)
$ npm install moment
npm http GET https://registry.npmjs.org/moment
npm http 200 https://registry.npmjs.org/moment
npm http GET https://registry.npmjs.org/moment/-/moment-2.4.0.tgz
npm http 200 https://registry.npmjs.org/moment/-/moment-2.4.0.tgz
moment@2.4.0 node_modules\moment

Geoff@GEOFF-LAPTOP /c:/Data/Dev/Packt/WindowsAzureMobileServices/Main/Game/Win8/T
ileTapper/TileTapper/Scripts/TileTapper/service (master|MERGING)
$
```

7. We can see a `node_modules` directory in our service folder.
8. Add the modules to the repository:

```
$ git add *
```

9. Now, we can edit the script we want (if you want to do it in the portal, skip to committing and pushing).
10. Use the `require` method to get a reference to the installed package and use it as needed. I created a level name using the following code:

```
var moment = require('moment');
var name = moment().format('YYMMDD dddd [Level]');

// Add level
var level = {
  name: name,
  cells: board,
  time: time
}

levelsTable.insert(level, {
  success: function(results) {
    console.log("Inserted level");
  },
  error: function(error) {
    console.error(error);
  }});

console.log("Level JSON: %j", level);
```

11. Commit the changes:


```
$ git commit -m "Added moment package and modified daily script"
```
12. Push the changes back:


```
$ git push origin master
```
13. Test if the script changes work with the installed module (look in the **LOGS** tab for errors).

Summary

We've seen that scripts are fantastic for customizing our services in order to manipulate data and do pretty much anything we can think of using external services and a third-party library with an NPM module. We've also learned how to pull and push our scripts using the Git version control and install NPM modules using Node.

We're not yet done with scripts either. In the next chapter, we will send different types of push notifications with the `push` object.

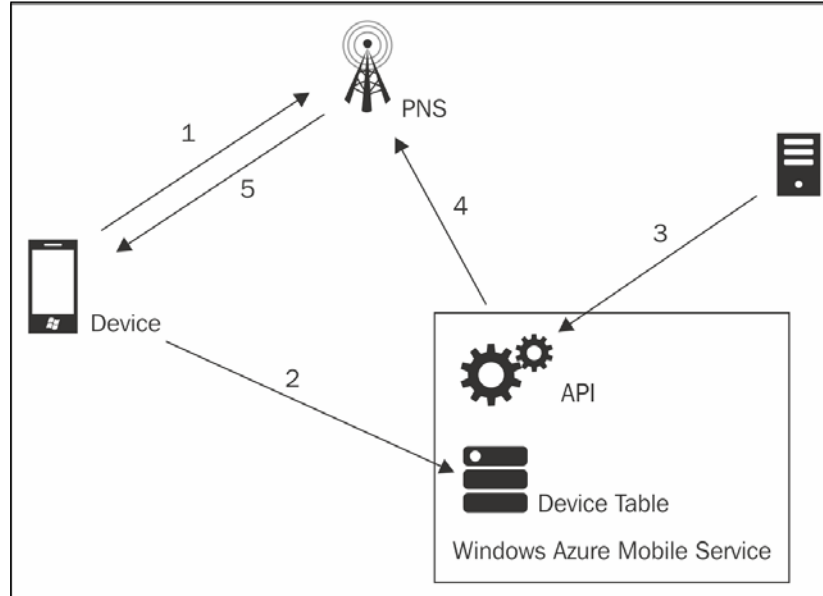
5

Implementing Push Notifications

Push Notifications allow us to expand our application's user experience outside the bounds of the app with live tile updates, toast notifications, and badges in Windows 8. Windows Azure Mobile Services makes it very easy for us to trigger notifications via **Windows Notifications Service (WNS)** (for Store apps), **Microsoft Push Notification Service (MPNS)** (for Windows Phone 8 apps), **Apple Notifications Service (ANS)**, and **Google Notifications Service (GCM)**. We're going to discuss how to configure Windows 8 and Windows Phone applications to allow notifications, send different types of notifications using scripts, and create a list of devices to manage our user's notification channels.

Understanding Push Notification Service flow

The following procedure illustrates **Push Notification Service (PNS)** flow from establishing a channel to receiving a notification:



1. The mobile device establishes a channel with the PNS and retrieves its handle (URI).
2. The device registers its handle with a backend service (in our case, a table in our Mobile Service).
3. A notification request can be made by another service, an admin system, and so on, which calls the backend service (in our case, an API).
4. The service makes a request to the correct PNS for every device handle.
5. The PNS notifies the device.

Setting up Windows Store apps

Visual Studio 2013 has a new wizard, which associates the app with the store in order to obtain a push notifications URI. Code is added to the app to interact with the service that will be updated to have a **Channels** table. This table has an `Insert` script to insert the channel and ping back a toast notification upon insert. The following procedure takes us through using the wizard to add a push channel to our app:

1. Right-click on the project, and then navigate to **Add | Push Notification**.
2. Follow the wizard and sign in to your store account (if you haven't got one, you will need to create one).
3. Reserve an app name and select it. Then, continue by clicking on **Next**.
4. Click on **Import Subscriptions...** and the **Import Windows Azure Subscriptions** dialog box will appear.
5. Click on **Download subscription file**. Your default browser will be launched and the subscriptions file will be automatically downloaded. If you are logged into the portal, this will happen automatically; otherwise, you'll be prompted to log in.
6. Once the subscription file is downloaded, browse to the downloaded file in the **Import Windows Azure Subscriptions** dialog box and click on **Import**.
7. Select the subscription you wish to use, click on **Next**, and then click on **Finish** in the final dialog box. In the **Output** window in Visual Studio, you should see something like the following:

```
Attempting to install 'WindowsAzure.MobileServices'
Successfully installed NuGet Package 'WindowsAzure.MobileServices'
Successfully added 'push.register.cs' to the project
Added field to the App class successfully
Initialization code was added successfully
Updated ToastCapable in the app manifest
Client Secret and Package SID were updated successfully on the
Windows Azure Mobile Services portal
The 'channels' table and 'insert.js' script file were created
successfully
Successfully updated application redirect domain
Done
```

We will now see a few things have been done to our project and service:

- The `Package.StoreAssociation.xml` file is added to link the project with the app on the store.
- `Package.appxmanifest` is updated with the store application identity.
- Add a `push.register.cs` class in `services\mobile services\[Your Service Name]`, which creates a push notifications channel and sends the details to our service.
- The server explorer launches and shows us our service with a newly created table named `channels`, with an `Insert` method that inserts or updates (if changed) our channel URI. Then, it sends us a toast notification to test that everything is working.

Run the app and check that the URI is inserted into the table. You will get a toast notification. Once you've done this, remove the `sendNotifications(item.channelUri)`; call and function from the `Insert` method. You can do this in Visual Studio via the **Server Explorer** console. I've modified the script further to make sure the item is always updated, so when we send push notifications, we can send them to URIs that have been recently updated so that we are targeting users who are actually using the application (channels actually expire after 30 days too, so it would be a waste of time trying to push to them). The following code details these modifications:

```
function insert(item, user, request)
{
    var ct = tables.getTable("channels");
    ct.where({ installationId: item.installationId }).read({
        success: function (results)
        {
            if (results.length > 0)
            {
                // always update so we get the updated date
                var existingItem = results[0];
                existingItem.channelUri = item.channelUri;
                ct.update(existingItem,
                {
                    success: function ()
                    {
                        request.respond(200, existingItem);
                    }
                });
            }
        }
    });
}
```

```

        else
        {
            // no matching installation, insert the record
            request.execute();
        }
    }
}
})
}

```

I've also modified the `UploadChannel` method in the app so that it uses a `Channel` model that has a `Platform` property. Therefore, we can now work out which PNS provider to use when we have multiple platforms using the service. The `UploadChannel` method also uses a new `InsertChannel` method in our `DataService` method (you can see the full code in the sample app). The following code details these modifications:

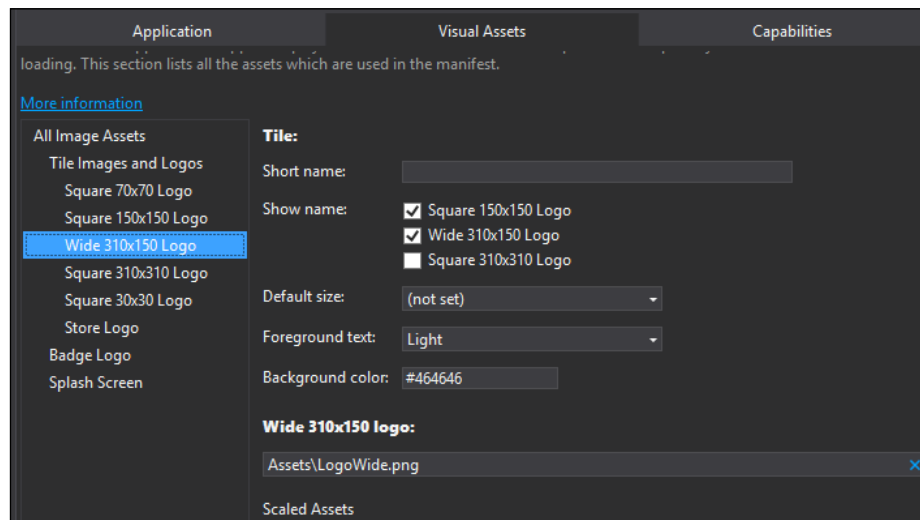
```

public async static void UploadChannel()
{
    var channel = await Windows.Networking.PushNotifications.
        PushNotificationChannelManager.
            CreatePushNotificationChannelForApplicationAsync();
    var token = Windows.System.Profile.
        HardwareIdentification.GetPackageSpecificToken(null);
    string installationId = Windows.Security.Cryptography.
        CryptographicBuffer.EncodeToBase64String(token.Id);
    try
    {
        var service = new DataService();
        await service.InsertChannel(new Channel()
        {
            ChannelUri = channel.Uri,
            InstallationId = installationId,
            Platform = "win8"
        });
    }
    catch (Exception ex)
    {
        System.Diagnostics.Debug.WriteLine(ex.ToString());
    }
}

```

Setting up tiles

To implement wide or large square tiles, we need to create the necessary assets and define them in the **Visual Assets** tab of the **Package.appxmanifest** editor. This is shown in the following screenshot:

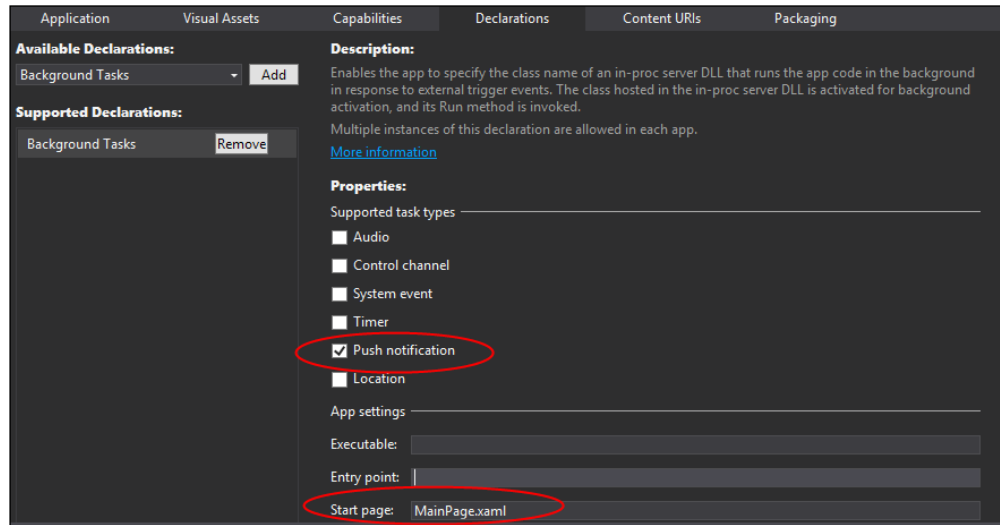


Setting up badges

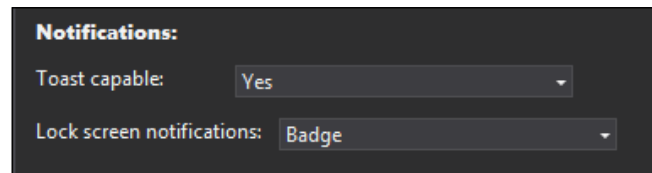
Windows Store apps support badge notifications as well as tile and toast. However, this requires a slightly different configuration. To implement badge notifications, we perform the following steps:

1. Create a 24 x 24 pixel PNG badge that can have opacity, but must use only white color.
2. Define the badge in the **Badge Logo** section of the **Visual Assets** tab of the **Package.appxmanifest** editor.

3. Add a **Background Tasks** declaration in the **Declarations** tab of the **Package.appxmanifest** editor, select **Push notification**, and enter a **Start page**, as shown in the following screenshot:



4. Finally, in the **Notifications** tab of the **Package.appxmanifest** editor, set **Lock screen notifications** to **Badge**. This is shown in the following screenshot:

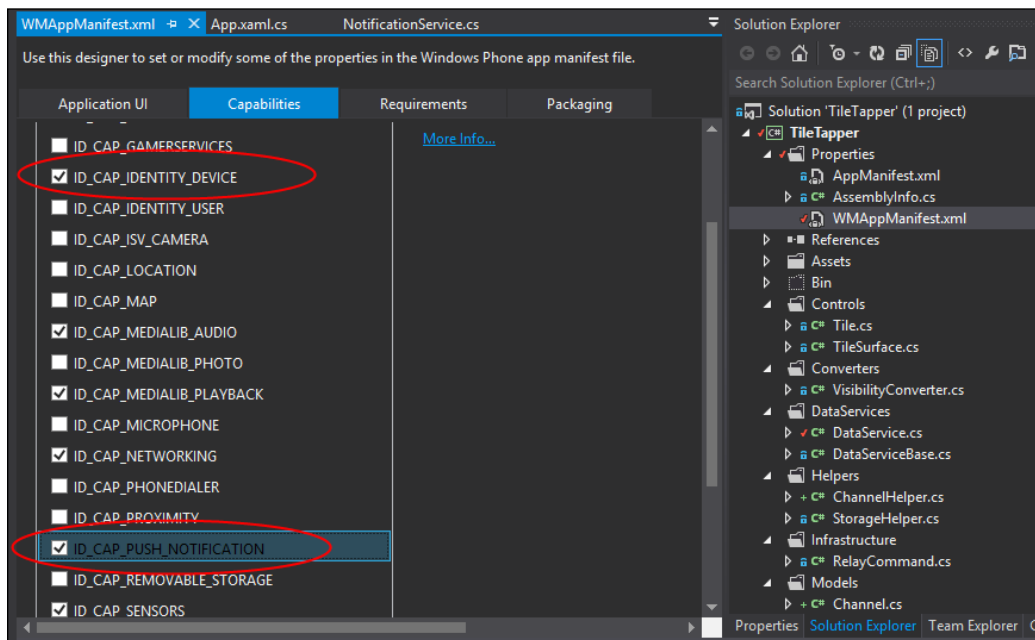


5. To see the badge notification working, you also need to add the app to the lock screen badge slots in **Lock Screen Applications | Change PC Settings | Lock Screen**.

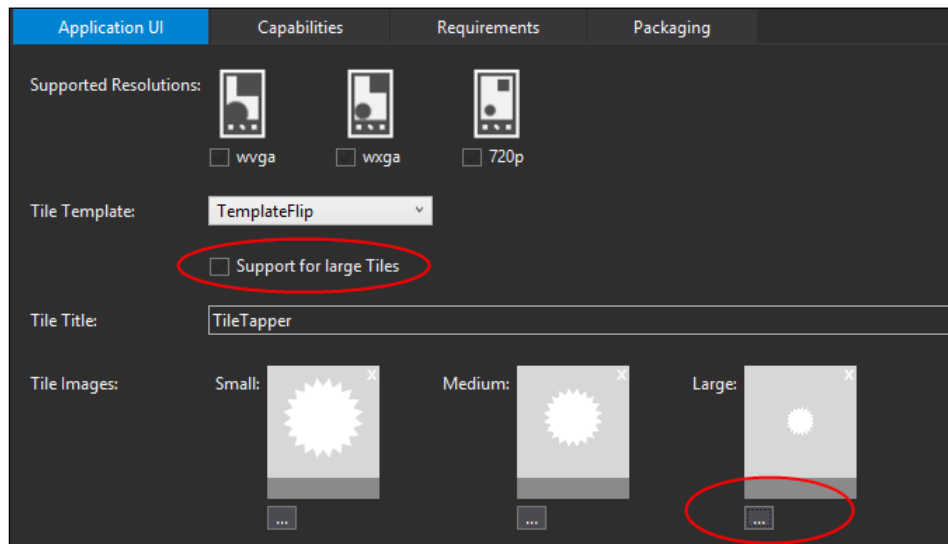
Setting up Windows Phone 8 apps

Visual Studio 2012 Express for Windows Phone doesn't have a fancy wizard like Visual Studio 2013 Express for Windows Store. So, we need to configure the channel and register it with the service manually. The following procedure sets up the notifications in the app by using the table that we created in the preceding *Setting up Windows Store apps* section:

1. Edit the `WMAppManifest.xml` file to enable `ID_CAP_IDENTITY_DEVICE`, which allows us to get a unique device ID for registering in the `Channels` table, and `ID_CAP_PUSH_NOTIFICATION`, which allows push notifications in the app. These options are available in the **Capabilities** tab, as shown in the following screenshot:



- To enable wide tiles, we need to check **Support for large Tiles** (you can't see the tick unless you hover over it, as there is apparently a theming issue in VS!) and pick the path of the wide tile we want to use (by default, there is one named `FlipCycleTileLarge.png` under `Tiles` in the `Assets` folder). This is shown in the following screenshot:



- Next, we need to add some code to get the push channel URI and send it to the service:

```
using Microsoft.Phone.Info;
using Microsoft.Phone.Notification;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Text;
using System.Threading.Tasks;
using TileTapper.DataServices;
using TileTapper.Models;

namespace TileTapper.Helpers
{
    public class ChannelHelper
    {
        // Singleton instance
        public static readonly ChannelHelper Default =
            new ChannelHelper();
    }
}
```



```
// Holds the push channel that is created or found
private HttpNotificationChannel _pushChannel;

// The name of our push channel
private readonly string CHANNEL_NAME =
    "TileTapperPushChannel";

private ChannelHelper() { }

public void SetupChannel()
{
    try
    {
        // Try to find the push channel
        this._pushChannel =
            HttpNotificationChannel.Find(CHANNEL_NAME);

        // If the channel was not found, then create a new
        // connection to the push service
        if (this._pushChannel == null )
        {
            this._pushChannel = new
                HttpNotificationChannel(CHANNEL_NAME);
            this.AttachEvents();
            this._pushChannel.Open();

            // Bind channel for Tile events
            this._pushChannel.BindToShellTile();

            // Bind channel for Toast events
            this._pushChannel.BindToShellToast();
        }
        else
            this.AttachEvents();
    }
    catch (Exception ex)
    {
        System.Diagnostics.Debug.WriteLine(ex.ToString());
    }
}

private void AttachEvents()
{
    // Register for all the events before attempting to
    // open the channel
    this._pushChannel.ChannelUriUpdated +=
        async (s, e) =>
        {
            // Register URI with service
        }
    }
}
```

```

        await this.Register();
    };

    this._pushChannel.ErrorOccurred += (s, e) =>
    {
        System.Diagnostics.Debug.WriteLine(e.ToString());
    };
}

private async Task Register()
{
    try
    {
        var service = new DataService();
        await service.InsertChannel(new Channel()
        {
            ChannelUri =
                this._pushChannel.ChannelUri.AbsoluteUri,
            InstallationId = this.GetDeviceUniqueName(),
            Platform = "wp8"
        });
    }
    catch (Exception ex)
    {
        System.Diagnostics.Debug.WriteLine(ex.ToString());
    }
}

// Note: to get a result requires
// ID_CAP_IDENTITY_DEVICE
// to be added to the capabilities of the WMAppManifest
// this will then warn users in marketplace
private byte[] GetDeviceUniqueID()
{
    byte[] result = null;
    object uniqueId;
    if (DeviceExtendedProperties.TryGetValue(
        "DeviceUniqueId", out uniqueId))
    {
        result = (byte[])uniqueId;
    }

    return result;
}

private string GetDeviceUniqueName()
{
    byte[] id = this.GetDeviceUniqueID();
    string idEnc = Encoding.Unicode.GetString(
        id, 0, id.Length);

```

```
        string deviceID = HttpUtility.UrlEncode(idEnc);  
        return deviceID;  
    }  
}
```

This is a singleton class that holds an instance of the `HttpNotificationChannel` object, so that channel URI changes can be captured and sent up to our service. The two methods at the end of the code snippet, `GetDeviceUniqueID` and `GetDeviceUniqueName`, will give a unique device identifier for the `channels` table.

4. Now that we have the code to manage the channel, we need to call the `SetupChannel` method in the `App.xaml.cs` launching method as shown in the following code snippet:

```
private void Application_Launching(  
    object sender, LaunchingEventArgs e)  
{  
    TileTapper.Helpers.ChannelHelper.Default.SetupChannel();  
}
```

Service scripts

In the `TileTapper` game, we send out notifications when a new level is created and when a new high score is submitted. We'll see how to send all the notification types (except raw; by all means do this if you need to in your application, but we're not going to discuss it now).

First, we'll look at a set of scripts which gets all the URIs from the `channels` table, which have been updated in the last 30 days so we know they are likely to be active and then sends notifications out to the correct PNS services depending on the platform type.

The `sendNotifications` function gets the channels from the `channels` table. Then, it loops through them, calling the `addToQueue` method that queues PNS task functions for each channel. We don't call the PNS methods in the `for` loop as they run asynchronously and would try to execute simultaneously, which would lead to many failures as the server can only make a limited number of HTTP requests concurrently. The following code demonstrates this:

```
// Queue of PNS functions  
var queue = [];
```

```

function sendNotifications(levelName)
{
    // Query channels updated in the last 30 days
    var sql = "SELECT channelUri, platform FROM channels WHERE
        updated >= DATEADD(Day, -30, GETDATE())";
    mssql.query(sql, {
        success: function(results)
        {
            // Because the PNS functions are asynchronous, we will loop
            // through channels
            // and add a set of functions to a function queue for each
            // channel so we can
            // process requests one at a time to save starving our
            // connections and failing
            for(var i = 0; i < results.length; i++)
            {
                addToQueue(results[i], levelName);
            }
            // Process first item
            dequeue();
        }
    });
}

```

The `addToQueue` function determines which notification functions are required, based on the platform type; and pushes a task function into the queue so that they can be called one at a time as they complete, as shown in the following code snippet:

```

// Wrap functions and enqueue
function addToQueue(channel, levelName)
{
    if(channel.platform == "win8")
    {
        queue.push(function() { sendMultiTileWns(
            channel.channelUri, levelName); });
        queue.push(function() { sendToastWns(
            channel.channelUri, levelName); });
        queue.push(function() { sendBadgeWns(
            channel.channelUri, levelName); });
    }
    else if(channel.platform == "wp8")
    {
        queue.push(function() { sendToastMpns(
            channel.channelUri, levelName); });
        queue.push(function() { sendTileMpns(
            channel.channelUri, levelName); });
    }
}

```

The `dequeue` method simply shifts a task function off the queue and calls it. As each function completes, it calls this function whether it succeeds or fails to empty the queue and process all PNS requests. The working of the `dequeue` method is shown in the following code snippet:

```
function dequeue()
{
    // Dequeue and execute
    if(queue.length > 0)
        (queue.shift())();
}
```

If a notification fails, we delete the channel registration from the table using the following function:

```
function deleteChannel(uri)
{
    var sql = "DELETE FROM channels WHERE channelUri = ' " + uri + "'";
    mssql.query(sql);
}
```

WNS scripts for Store apps

WNS supports the following notifications:

- `sendTile`
- `sendToast`
- `sendBadge`
- `sendRaw`
- `send`



`sendTile` and `sendToast` have a template-specific suffix to define the payload type.

WNS doesn't support tile templates with multiple tile sizes. So, we can use the `send` method to stick multiple tile bindings together and update more than one tile. There's a full reference available at <http://msdn.microsoft.com/en-us/library/windowsazure/jj860484.aspx>.

Sending toast notifications

The following function sends a toast notification using the `sendToastText04` method:

```
function sendToastWns(uri, name)
{
    // Send wns push for store apps
    push.wns.sendToastText04(uri, {
        text1: "TileTapper",
        text2: "New level available",
        text3: name
    }, {
        success: function(pushResponse)
        {
            console.log("Sent push toast WNS:", pushResponse);
            dequeue();
        },
        error: function(error)
        {
            console.error(error);
            deleteChannel(uri);
            dequeue();
        }
    });
}
```

Sending tile notifications

The following function sends a tile notification using the `sendTileSquareText01` method:

```
function sendTileWns(uri, name)
{
    // Send wns push for store apps
    push.wns.sendTileSquareText01(uri, {
        text1: "TileTapper",
        text2: "New level available",
        text3: name
    }, {
        success: function(pushResponse)
        {
            console.log("Sent push toast WNS:", pushResponse);
            dequeue();
        },
        error: function(error)
        {

```

```
        console.error(error);
        deleteChannel(uri);
        dequeue();
    }
    });
}
```

Sending multiple tiles

The following function sends a tile notification using multiple bindings that are defined using the raw XML templates. It gives us the benefit of sending multiple tile templates in one request, rather than sending them individually.

```
function sendMultiTileWns(uri, name)
{
    // Send wns push for store apps
    push.wns.send(uri,
        "<tile>" +
        "<visual version='2'>" +
        "<binding template =
            'TileSquare150x150Text01' fallback='TileSquareText01'>" +
            "<text id='1'>TileTapper</text>" +
            "<text id='2'>New level available</text>" +
            "<text id='3'>" + name + "</text>" +
        "</binding>" +
        "<binding template =
            'TileWide310x150Text01' fallback='TileWideText01'>" +
            "<text id='1'>TileTapper</text>" +
            "<text id='2'>New level available</text>" +
            "<text id='3'>" + name + "</text>" +
        "</binding>" +
        "</visual>" +
        "</tile>",
        "wns/tile", {
            success: function(pushResponse)
            {
                console.log("Sent push toast WNS:", pushResponse);
                dequeue();
            },
            error: function(error)
            {
                console.error(error);
                deleteChannel(uri);
                dequeue();
            }
        }
    );
}
```

Sending badge notifications

The following function sends an alert badge notification using the `sendBadge` method:

```
function sendBadgeWns(uri, name)
{
    // Send wns push for store apps
    push.wns.sendBadge(uri, "alert", {
        success: function(pushResponse)
        {
            console.log("Sent push toast WNS:", pushResponse);
            dequeue();
        },
        error: function(error)
        {
            console.error(error);
            deleteChannel(uri);
            dequeue();
        }
    });
}
```

MPNS scripts for Windows Phone apps

MPNS supports the following notifications:

- `sendFlipTile`
- `sendTile`
- `sendToast`
- `sendRaw`

There's a full reference available at <http://msdn.microsoft.com/en-us/library/windowsazure/jj871025.aspx>.

Sending toast notifications

The following function sends a toast notification using the `sendToast` method:

```
function sendToastMpns(uri, name)
{
    // Send wns push for store apps
    // We can add a param object to pass params to a certain page:
    // param: "NewPage.xaml?item=5"
    push.mpns.sendToast(uri, {
        text1: "TileTapper - New level available",
        text2: name
    }, {
```



```
        success: function(pushResponse)
        {
            console.log("Sent push toast WNS:", pushResponse);
            dequeue();
        },
        error: function(error)
        {
            console.error(error);
            deleteChannel(uri);
            dequeue();
        }
    });
}
```

Sending tile notifications

The following function sends a tile notification using the `sendFlipTile` method:

```
function sendTileMpns(uri, name)
{
    // Send wns push for store apps
    // We can add a param object to pass params to a certain page:
    // param: "NewPage.xaml?item=5"
    push.mpns.sendFlipTile(uri, {
        backTitle: "TileTapper - New level available",
        backContent: name
    }, {
        success: function(pushResponse)
        {
            console.log("Sent push toast WNS:", pushResponse);
            dequeue();
        },
        error: function(error)
        {
            console.error(error);
            deleteChannel(uri);
            dequeue();
        }
    });
}
```

Summary

In this chapter, we've covered setting up our Windows 8 and Windows Phone 8 applications to receive different notification types. We have also worked on the service to send different notifications from the WNS and MPNS notifications service.

Tiles and toast notifications are big subjects as there are a plethora of templates on each platform. So, it's worth having a good look at the documentation to help you choose the right templates.

In the next chapter, we're going to build on what we've learned here with the Notifications Hub, which provides us with a different, more scalable mechanism for managing push notifications.

6

Scaling Up with the Notifications Hub

The PNS facilities in Azure Mobile Services are great, but Azure has a more scalable solution, available to us from the Service Bus group of services.

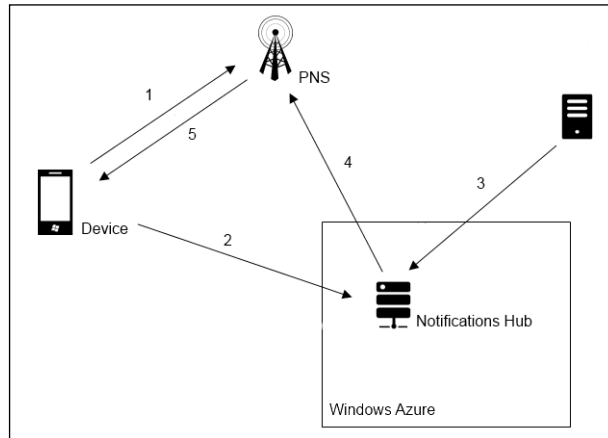
The Notifications Hub has the following benefits over push notifications:

- Manages device URI handles for us
- Only requires a single request from the backend to broadcast notifications
- Offers generic notifications across all platforms as well as native notification types
- Tags to allow users to filter notifications
- Provides language support

The Notifications Hub flow is described as follows:

1. The mobile device establishes a channel with the PNS and retrieves its URI handle.
2. The device registers with the Notifications Hub.
3. A notification request is made by another service or an admin system to the hub.

4. The service makes a request for every device handle to the correct PNS.
5. The PNS notifies the device.



The main drawback of using the hub over Mobile Services push notifications is the separate pricing model. You get 1,00,000 pushes per month on 500 devices for free. On unlimited devices, you get 1 million pushes and 5 million pushes (per unit) for basic and standard subscriptions, respectively.

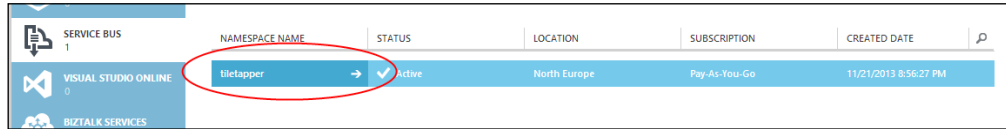
Configuring the Hub

First, we're going to configure our Notifications Hub in the Service Bus Portal. The steps are as follows:

1. In the Azure Portal, select **SERVICE BUS** from the left menu.
2. Click on **CREATE A NAMESPACE**.
3. Enter a name, select a region (pick the same one as you used for the database and mobile service), and choose a subscription:

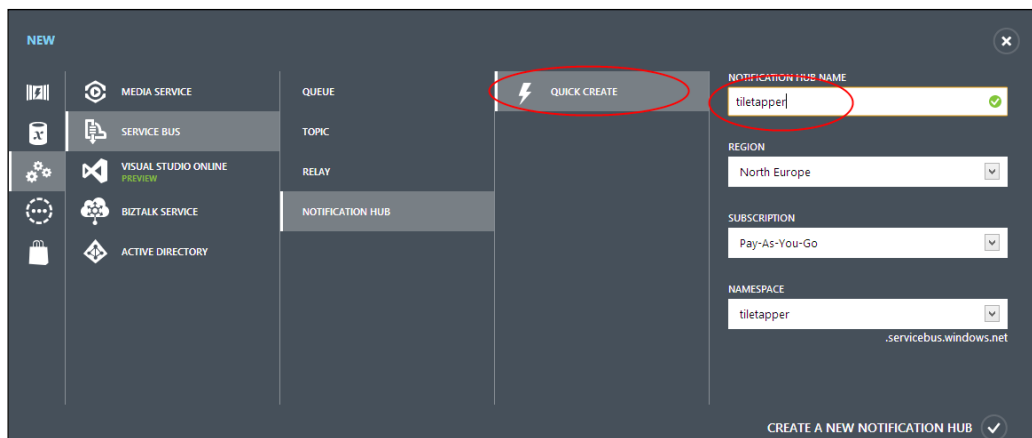
```
CREATE A NAMESPACE
Add a new namespace
NAMESPACE NAME: tiletapper
REGION: North Europe
SUBSCRIPTION: Pay-As-You-Go
```

- Click on the newly created namespace to enter the **SERVICE BUS** Portal:



	NAMESPACE NAME	STATUS	LOCATION	SUBSCRIPTION	CREATED DATE
	tiletapper	Active	North Europe	Pay-As-You-Go	11/21/2013 8:56:27 PM

- Select **NOTIFICATION HUB** from the menu.
- Click on **CREATE A NEW NOTIFICATIONS HUB**.
- From the pop-up menu, click on **QUICK CREATE**:



NEW

MEDIA SERVICE

SERVICE BUS

VISUAL STUDIO ONLINE

BIZTALK SERVICE

ACTIVE DIRECTORY

QUEUE

TOPIC

RELAY

NOTIFICATION HUB

QUICK CREATE

NOTIFICATION HUB NAME

tiletapper

REGION

North Europe

SUBSCRIPTION

Pay-As-You-Go

NAMESPACE

tiletapper

.servicebus.windows.net

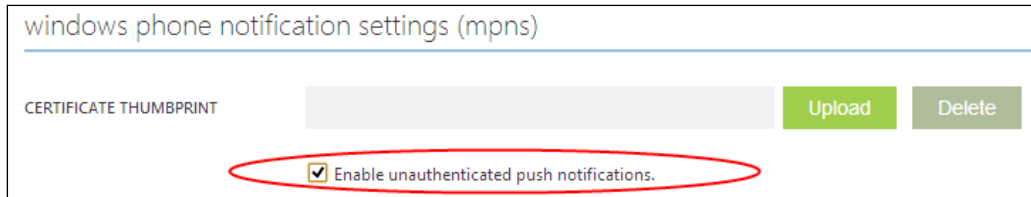
CREATE A NEW NOTIFICATION HUB

- Enter a name in the **NOTIFICATION HUB NAME** field and click on the **CREATE A NEW NOTIFICATION HUB** tick.
- For Windows Store apps, under the Mobile Services Portal's **CONFIGURE** tab, copy the **CLIENT SECRET** and **PACKAGE SID** keys from the **PUSH** tab in the Mobile Services Portal, created when we configured push notifications in the previous chapter. Paste them into the **windows phone notification settings** section under the **CONFIGURE** tab.



Note that, at the time of writing this, they were in the opposite order!

10. For Windows Phone 8 apps, under the **CONFIGURE** tab, check the **Enable unauthenticated push notifications** checkbox:



windows phone notification settings (mpns)

CERTIFICATE THUMBPRINT Upload Delete

☒ Enable unauthenticated push notifications.

If you have obtained a MPNS certificate, you can use it here to get un-throttled authenticated notifications.

Setting up Windows Store and Windows Phone 8 apps

The following procedure sets up hub notifications in Windows 8 and Windows Phone 8 apps:

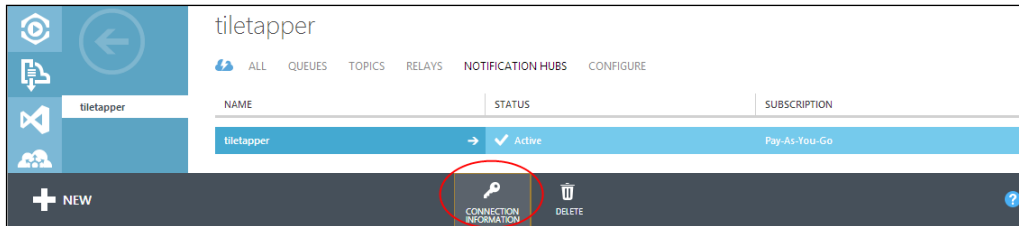
1. Install the `WindowsAzure.Messaging.Managed` NuGet package by entering the following command in the Package Manager Console:
Install-Package WindowsAzure.Messaging.Managed
2. Add the following namespace references to the `ChannelHelper` (Windows Phone 8) or `TileTapperPush` (Windows 8) class we created in the previous chapter:

```
using Microsoft.Phone.Notification;  
using Microsoft.WindowsAzure.Messaging;
```

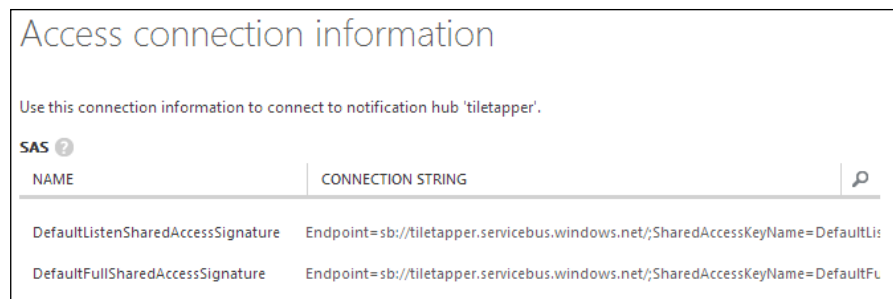
3. Add the following constants to the top of the class and change the `HUB_NAME` constant to your hub name:

```
private readonly string HUB_NAME = "tiletapper";  
private readonly string CONNECTION_STRING = "Endpoint=sb://  
tiletapper.servicebus.windows.t/;SharedAccessKeyName=DefaultListen  
SharedAccessSignature;SharedAccessKey=/xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx  
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx=";
```

4. In the **NOTIFICATION HUBS** Portal, click on the **CONNECTION INFORMATION** button on the toolbar to see the connection strings:



5. Copy the **DefaultListenSharedAccessSignature** string (use the copy button) and paste it into the **CONNECTION STRING** constant:



6. Add the following to the `ChannelHelper.Register` (Windows Phone 8) or `TileTapperPush.UploadChannel` (Windows 8) task:


```
// Register with hub
var hub = new NotificationHub(this.HUB_NAME, this.CONNECTION_STRING);
var result = await hub.RegisterNativeAsync(this._pushChannel.ChannelUri.AbsoluteUri);
```
7. For Windows Phone 8, uninstall the app, then re-deploy and run it to get the channel to refresh and register with the hub.
8. It may take a few minutes once the `RegisterNativeAsync` has been called for the channel to register and receive notifications.

Calling the hub from scripts

In the previous chapter, we talked about sending push notifications from our scripts using the PNS libraries. There isn't a built-in library for the Notifications Hub, but remember when we were looking at using NPM modules in our scripts? Well, we can pull in a reference to the Azure SDK for a Node NPM package, which is preinstalled in our service, so we don't even need to install it! The SDK is open source and can be found on GitHub at <https://github.com/WindowsAzure/azure-sdk-for-node>. It can be useful if you are having trouble finding examples of how to do certain things because you can look at the code.

If you remember in the previous chapter, we had to maintain a table of channels, then loop through the table, determine which provider to use, build a queue of PNS functions for each channel URI, and call them one at a time! Well, for the Notifications hub, this couldn't be simpler as we just need to make a single call to the hub for each notification type we want to send:

```
function sendAllHubNotifications(levelName)
{
    sendToastHubMpns("TileTapper - New level available", levelName,
null);
    sendTileHubMpns("TileTapper - New level available", levelName,
null);

    sendToastHubWns("TileTapper", "New level available", levelName,
null);
    sendTileHubWns("TileTapper", "New level available", levelName,
null);
    sendBadgeHubWns("alert", null);
}
```

All PNS methods have prototypes similar to this (from SDK code):

```
MpnsService.prototype.send = function (tags, payload, targetName,
notificationClass, optionsOrCallback, callback)
```

Most of the parameters are self-explanatory; however, the `notificationClass` controls the batching interval (you can read more on this at <http://msdn.microsoft.com/en-us/library/hh221551.aspx>).

All the scripts shown next use the same constants:

```
var CONNECTION_STRING = "Endpoint=sb://tiletapper.servicebus.windows.
net/;SharedAccessKeyName=DefaultFullSharedAccessSignature;SharedAccess
Key=XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX=";
var HUB_NAME = "tiletapper";
```

Use `DefaultFullSharedAccessSignature` from the Notifications Hub Portal.

Creating WNS scripts (for Store apps)

We've already talked about the different notification types and templates, so we'll just look at the code.

Sending toast notifications

The following function sends a WNS toast notification:

```
function sendToastHubWns(text1, text2, text3, tagExpression)
{
  var azure = require("azure");
  var notificationHubService = azure.createNotificationHubService(HUB_NAME, CONNECTION_STRING);

  var toast = "<toast>" +
    "<visual>" +
      "<binding template=\"ToastText04\">" +
        "<text id=\"1\">" + text1 + "</text>" +
        "<text id=\"2\">" + text2 + "</text>" +
        "<text id=\"3\">" + text3 + "</text>" +
        "</binding>" +
      "</visual>" +
    "</toast>";

  notificationHubService.wns.send(tagExpression, toast, "wns/toast", 2,
  function(error) {
    if (error) {
      console.error(error);
    }
  });
}
```

Sending tile notifications

The following function sends a WNS tile notification:

```
function sendTileHubWns(text1, text2, text3, tagExpression)
{
  var azure = require("azure");
  var notificationHubService = azure.createNotificationHubService(HUB_NAME, CONNECTION_STRING);

  var tile = "<tile>" +
    "<visual>" +
      "<binding template=\"TileSquareText01\">" +
```

```
        "<text id=\"1\">" + text1 + "</text>" +
        "<text id=\"2\">" + text2 + "</text>" +
        "<text id=\"3\">" + text3 + "</text>" +
        "</binding>" +
        "</visual>" +
        "</tile>";

notificationHubService.wns.send(tagExpression, tile, "wns/tile", 1,
function(error) {
    if (error) {
        console.error(error);
    }
}));
}
```

Sending badge notifications

The following function sends a WNS badge notification:

```
function sendBadgeHubWns(value, tagExpression)
{
    var azure = require("azure");
    var notificationHubService = azure.createNotificationHubService
        (HUB_NAME, CONNECTION_STRING);

    var badge = "<badge value=\"" + value + "\" />";

    notificationHubService.wns.send(tagExpression, badge, "wns/badge", 2,
function(error) {
    if (error) {
        console.error(error);
    }
}));
}
```

Creating MPNS scripts (for Windows Phone 8 apps)

Again, we've already talked about the different notification types and templates, so we'll just look at the code.

Sending toast notifications

The following function sends a MPNS toast notification:

```
function sendToastHubMpns(text1, text2, tagExpression)
{
  var azure = require("azure");
  var notificationHubService = azure.createNotificationHubService
    (HUB_NAME, CONNECTION_STRING);

  var toast = "<?xml version='1.0' encoding='utf-8'?" +
    "<wp:Notification xmlns:wp='WPNotification'" +
    "<wp:Toast" +
    "<wp:Text1" + text1 + "</wp:Text1" +
    "<wp:Text2" + text2 + "</wp:Text2" +
    "</wp:Toast" +
    "</wp:Notification";

  notificationHubService.mpns.send(tagExpression, toast, "toast", 2,
    function(error) {
      if (error) {
        console.error(error);
      }
    });
}
```

Sending tile notifications

The following function sends a MPNS tile notification:

```
function sendTileHubMpns(backTitle, backContent, tagExpression)
{
  var azure = require("azure");
  var notificationHubService = azure.createNotificationHubService
    (HUB_NAME, CONNECTION_STRING);

  var tile = "<?xml version='1.0' encoding='utf-8'?" +
    "<wp:Notification xmlns:wp='WPNotification' Version='2.0'" +
    "<wp:Tile Template='FlipTile'" +
    "<wp:BackTitle" + backTitle + "</wp:BackTitle" +
    "<wp:BackContent" + backContent + "</wp:BackContent" +
    "<wp:WideBackContent" + backContent + "</wp:WideBackContent" +
    "</wp:Tile" +
    "</wp:Notification";

  notificationHubService.mpns.send(tagExpression, tile, "token", 1,
    function(error) {
```

```
    if (error) {  
        console.error(error);  
    }  
}
```

Backend services

Similar to calling the Notifications Hub from our scripts, we can call it from any backend services we may have for generating app content and so on. To do this in a .NET application, follow this procedure:

1. Install the Windows Azure Service Bus SDK NuGet package by typing the following into the **Package Manager Console**:

```
Install-Package WindowsAzure.ServiceBus
```

2. Add the following namespace:

```
using Microsoft.ServiceBus.Notifications;
```

3. Add constants for the connection string and hub name:

```
private const string CONNECTION_STRING = "Endpoint=sb://  
tiletapper.servicebus.windows.net/;SharedAccessKeyName=DefaultFull  
SharedAccessSignature;SharedAccessKey=xxxxxxxxxxxxxxxxxxxxxxxxxxxx  
xxxxxxxxxx=";  
private const string HUB_NAME = "tiletapper";
```

4. We can then send notifications shown as follows:

```
private async Task SendToastHubMpns(string text1, string text2,  
string tagExpression)  
{  
    NotificationHubClient hub = NotificationHubClient.CreateClient  
FromConnectionString(CONNECTION_STRING, HUB_NAME);  
    string toast = "<?xml version='1.0' encoding='utf-8'?" +  
        "<wp:Notification xmlns:wp='WPNotification'" +  
        "<wp:Toast'" +  
        "<wp:Text1'" + text1 + "</wp:Text1'" +  
        "<wp:Text2'" + text2 + "</wp:Text2'" +  
        "</wp:Toast'" +  
        "</wp:Notification'";  
    var result = await hub.SendMpnsNativeNotificationAsync(toast,  
tagExpression);  
}
```

We'll not go into all the different notification types again as the payloads are very similar to the Node version.

Targeting audience using tags

The Notifications Hub has a concept of tagging notifications, whereby a user can pick the types of notifications they are interested in. The app registers these as tags and the backend service sends out tagged notifications, so users only get notifications they want to receive.

In the TileTapper game, I created a TagHelper class that allows the settings page to control notifications that the user wants to receive (via the view model):

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace TileTapper.Helpers
{
    public class TagHelper
    {
        private const string TILE_HIGH_SCORE = "TILE_HIGH_SCORE";
        private const string TOAST_HIGH_SCORE = "TOAST_HIGH_SCORE";
        private const string TILE_LEVEL = "TILE_LEVEL";
        private const string TOAST_LEVEL = "TOAST_LEVEL";

        // Singleton instance
        public static readonly TagHelper Default = new TagHelper();

        private TagHelper()
        {
        }

        public bool IsTileHighScoreEnabled
        {
            get { return StorageHelper.GetSetting<bool>(TILE_HIGH_SCORE); }
            set { StorageHelper.StoreSetting(TILE_HIGH_SCORE, value, true); }
        }

        public bool IsToastHighScoreEnabled
        {

```

```
        get { return StorageHelper.GetSetting<bool>(TOAST_HIGH_SCORE); }
        set { StorageHelper.StoreSetting(TOAST_HIGH_SCORE, value, true); }
    }

    public bool IsTileLevelEnabled
    {
        get { return StorageHelper.GetSetting<bool>(TILE_LEVEL); }
        set { StorageHelper.StoreSetting(TILE_LEVEL, value, true); }
    }

    public bool IsToastLevelEnabled
    {
        get { return StorageHelper.GetSetting<bool>(TOAST_LEVEL); }
        set { StorageHelper.StoreSetting(TOAST_LEVEL, value, true); }
    }

    public IEnumerable<string> GetTags()
    {
        var tags = new List<string>();

        if (this.IsTileHighScoreEnabled)
            tags.Add(TILE_HIGH_SCORE);

        if (this.IsToastHighScoreEnabled)
            tags.Add(TOAST_HIGH_SCORE);

        if (this.IsTileLevelEnabled)
            tags.Add(TILE_LEVEL);

        if (this.IsToastLevelEnabled)
            tags.Add(TOAST_LEVEL);

        return tags;
    }
}
```

When these change we need to re-register our channel with the hub with the list of tags like this:

```
// Register with hub
var tags = TagHelper.Default.GetTags();
var hub = new NotificationHub(this.HUB_NAME, this.CONNECTION_STRING);
var result = await hub.RegisterNativeAsync(this._pushChannel.
ChannelUri.AbsoluteUri, tags);
```

Now, when we send notifications in our service, we can add tags to the requests as follows:

```
// Hub functions
function sendAllHubNotifications(levelName)
{
    sendToastHubMpnns("TileTapper - New level available", levelName,
"TOAST_LEVEL");
    sendTileHubMpnns("TileTapper - New level available", levelName,
"TOAST_LEVEL");

    sendToastHubWns("TileTapper", "New level available", levelName,
"TOAST_LEVEL");
    sendTileHubWns("TileTapper", "New level available", levelName,
"TOAST_LEVEL");
    sendBadgeHubWns("alert", "BADGE_LEVEL");
}
```

At the time of writing this, there seems to be an issue with this working on Windows Phone; however, it works fine on Windows 8.

Summary

In this chapter, we've seen how using the Notifications Hub can save us a lot of work managing push notifications. It is probably a better choice over the built-in push notifications support in the service.

The hub also offers a really good template feature that allows apps to register templates for notification categories that the user is interested in, with just a single notification request required on the server side. This is very powerful as we can target multiple platforms with one request and provide localization support. Unfortunately, we do not have the time to look at this now, but there are some good references:

- <http://msdn.microsoft.com/en-us/library/windowsazure/dn530748.aspx>
- <http://www.windowsazure.com/en-us/manage/services/notification-hubs/breaking-news-localized-dotnet/>

Next is the final chapter in which we are going to look at tying up everything we've learned so far in the book and getting our apps ready for the store!

7

Best Practices for Web-connected Apps

In this final chapter, we're going to look at what we need to do to prepare our apps for store certification and to improve **user experience (UX)** with respect to network connectivity and push notifications.

There are certain criteria your app must meet to be published on the store and guidelines to help create better UX. Windows Store app guidelines are pretty comprehensive and cover everything needed for Windows Phone apps too. There are some specifics that need particular notice for any web-connected app and apps that implement push notifications.

App certification requirements for the Windows Store

It's worth reading through the *App certification requirements for the Windows Store* section for general requirements of the applications (at the time of writing this, the document version is 4.7, October 17, 2013), available at <http://msdn.microsoft.com/en-us/library/windows/apps/hh694083.aspx>.

In particular, the following apply to this subject:

4.1.1 Your app must have a privacy statement if it is network-capable

If your app has the technical ability to transmit data, you must maintain a privacy policy. You must provide access to your privacy policy in the Description page of your app, as well as in the app's settings as displayed in the Windows Settings charm.

App capability declarations that make your app network-capable include: `internetClient`, `internetClientServer` and `privateNetworkClientServer`.

Your privacy policy must inform users of the personal information transmitted by your app and how that information is used, stored, secured and disclosed, and describe the controls that users have over the use and sharing of their information, how they may access their information, and it must comply with applicable laws and regulations.

4.2 Your app must respect system settings for notifications and remain functional when they are disabled

This includes the presentation of ads and notifications to the customer, which must also be consistent with the customer's preferences, whether the notifications are provided by the Windows Push Notification Service or any other service. If a customer disables the notification function, either on an app-specific or a system-wide basis, your app must remain functional.

This means, we need to provide a privacy policy and make sure our apps function when notifications are disabled either through the operating system or from our app. It's a good idea to allow users to control notifications from the app. Let's take a look at how to do it.

UX guidelines

The user guidelines are a good resource for helping us create user-friendly applications:

<http://msdn.microsoft.com/en-us/library/windows/apps/hh465424.aspx>

Of particular interest is the *Guidelines for connection usage data* section available at <http://msdn.microsoft.com/en-us/library/windows/apps/hh452974.aspx>, specifically the following table:

Network Cost Type	Recommended App Behavior
Unrestricted	Use the network connection freely.
Variable/ Approaching Data Cap	<ul style="list-style-type: none"> • Delay or schedule lower priority operations until an unrestricted network is available. • When streaming content to a user, such as a movie or a video, use a lower bit-rate. For example, if your app is streaming HD-Quality video, stream Standard Definition when on a metered network. • Use less bandwidth. For example, switching to header-only mode when receiving emails. • Use the network less frequently. An example solution is to reduce the frequency of any polling operations for syndicating news feeds, refreshing content for a website, or getting web notifications. • Allow users to explicitly choose to stream HD-Quality video, retrieve full emails, download lower priority updates, etc., rather than doing so by default. • Explicitly ask for user permission prior to using the network.
Unknown	If the network cost type is unknown, then treat it as an unrestricted network.

Also, read the *Guidelines for push notifications (Windows Store apps)* section at <http://msdn.microsoft.com/en-us/library/windows/apps/hh761462.aspx>, with particular attention to the following points:

- Respect your user's battery life
- Do not use push notifications for spam or with malicious intent
- Be aware that WNS has no delivery guarantees
- Do not send confidential or sensitive data through push notifications
- Keep your app server credentials a secret

The following guidelines are also worth referring to:

- **Guidelines for toast notifications (Windows Store apps):** <http://msdn.microsoft.com/en-us/library/windows/apps/hh465391.aspx>
- **Guidelines for tiles and badges (Windows Store apps):** <http://msdn.microsoft.com/en-us/library/windows/apps/hh465403.aspx>

Implementing a privacy policy

When we create applications that connect to Internet services, we need to provide a privacy policy that can be viewed in the app and that is needed for a Windows Store app submission.

A privacy policy can be embedded in the app, but it's easier to have one on your website (you need a website when you fill in your app's details on the store), and then put a link on the settings page. There are a number of free policy generators that can be used to quickly produce a policy. There's a good overview of different generators available at <http://www.applicationprivacy.org/do-tools/privacy-policy-generator/>.

Checking the network connection

Before we connect to our services or try and authenticate the user, we can check if the device actually has the capability of making a request using the `NetworkInterface.GetIsNetworkAvailable` method. We can also examine the cost involved using the `NetworkInformation.GetInternetConnectionProfile` method to determine whether we should warn the user about potentially high data costs (these are the same for Windows 8 and Windows Phone 8). This is shown in the following code snippet:

```
using System;
using System.Net.NetworkInformation;
using System.Threading.Tasks;
using Windows.Networking.Connectivity;
using Windows.UI.Popups;

namespace TileTapper.Helpers
{
    public class NetworkHelper
    {
        public async static Task<bool> CheckAvailablity()
        {
            // Check network availability
            if (!NetworkInterface.GetIsNetworkAvailable())
                return false;

            // Check cost
            var cp = NetworkInformation.GetInternetConnectionProfile();
            var cost = cp.GetConnectionCost();

            if (cost.NetworkCostType == NetworkCostType.Unrestricted
                || cost.NetworkCostType == NetworkCostType.Unknown)
```

```

        return true;

    else if ((cost.NetworkCostType == NetworkCostType.Fixed
        || cost.NetworkCostType == NetworkCostType.Variable
        ) && (!cost.OverDataLimit && !cost.Roaming))
        return true;

    // If none of the above criteria are met, ask user if they
    // wish to continue
    bool available = false;
    var title = "Network Usage Warning";
    var content = "The application needs to get data over the
        internet, but your current network cost may be high. Do
        you wish to proceed?";
    var md = new MessageDialog(content, title);
    md.Commands.Add(new UICommand("Yes", (e) =>
        { available = true; }));
    md.Commands.Add(new UICommand("No"));
    md.CancelCommandIndex = 1;
    md.DefaultCommandIndex = 0;

    await md.ShowAsync();

    return available;
}
}
}

```

For Windows Phone, the `MessageDialog` class is replaced with a `MessageBox` class, otherwise, the methods are the same.

It is also possible to detect when the connection changes using the `NetworkAddressChanged` event. In the `TileTapper` game, the constructor hooks the event and then checks the network in the `MainVM` constructor. If the network becomes available and the game has not initialized, this is then done:

```

public MainVM()
{
    // Constructor code removed for brevity

    // Detect network changes and check current state
    System.Net.NetworkInformation.NetworkChange.
        NetworkAddressChanged += (s, e) => CheckNetwork();
    this.CheckNetwork();
}

```

```
private async void CheckNetwork()
{
    // Check network is available
    if (!await NetworkHelper.CheckAvailablity())
        this.IsNetworkOverlayVisible = true;
    else
    {
        // Initialise if required
        if (!this._isInitialised)
        {
            this.Initialise();
            this._isInitialised = true;
        }

        this.IsNetworkOverlayVisible = false;
    }
}
```

Managing notifications settings

For the Windows 8 app, we will use the `TagHelper` class discussed in *Chapter 6, Scaling Up with the Notifications Hub*, to manage the types of notifications that the user is interested in. For the Windows Phone app, a new `SettingsHelper` singleton class is used, which just manages a single property accessed by the view model and the `ChannelHelper` class. This is shown in the following code snippet:

```
namespace TileTapper.Helpers
{
    public class SettingsHelper
    {
        private const string PUSH_ENABLED = "PUSH_ENABLED";

        // Singleton instance
        public static readonly SettingsHelper Default =
            new SettingsHelper();

        private SettingsHelper() { }

        public bool IsPushEnabled
        {
            get {return StorageHelper.GetSetting<bool>(PUSH_ENABLED);}
            set {StorageHelper.StoreSetting(PUSH_ENABLED, value, true);}
        }
    }
}
```

The `ChannelHelper` class is modified to close and dispose the channel and unregister with the service and hub, if needed. This is shown in the following code snippet:

```
public async Task SetupChannel()
{
    try
    {
        bool attach = false;
        // Try to find the push channel
        if (this._pushChannel == null)
        {
            attach = true;
            this._pushChannel =
                HttpNotificationChannel.Find(CHANNEL_NAME);
        }

        // Check if user has enabled
        bool enabled = SettingsHelper.Default.IsPushEnabled;

        // If the channel was not found, then create a new connection
        // to the push service.
        if (this._pushChannel == null && enabled)
        {
            this._pushChannel =
                new HttpNotificationChannel(CHANNEL_NAME);
            this.AttachEvents();
            this._pushChannel.Open();

            // Bind channel for Tile events.
            this._pushChannel.BindToShellTile();

            // Bind channel for Toast events
            this._pushChannel.BindToShellToast();
        }
        // If channel was found but not required, close it
        else if (this._pushChannel != null && !enabled)
        {
            await this.UnRegister();

            this._pushChannel.Close();
            this._pushChannel.Dispose();
            this._pushChannel = null;
        }
    }
}
```



```
        // Channel is found and needed so just attach
        else if (this._pushChannel != null && enabled && attach)
            this.AttachEvents();
    }
    catch (Exception ex)
    {
        System.Diagnostics.Debug.WriteLine(ex.ToString());
    }
}

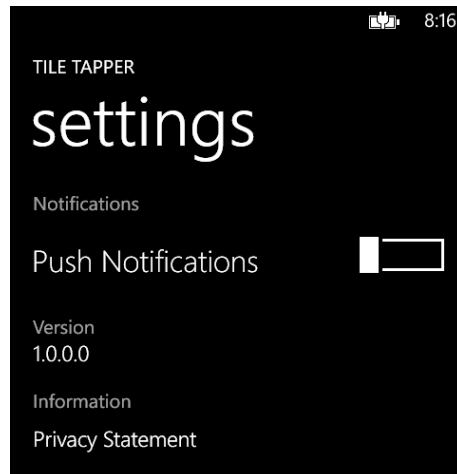
public async Task UnRegister()
{
    try
    {
        // UnRegister with service
        var service = new DataService();
        await service.DeleteChannel(this.GetDeviceUniqueName());

        // UnRegister with hub
        var hub = new NotificationHub(this.HUB_NAME,
            this.CONNECTION_STRING);
        await hub.UnregisterAllAsync(
            this._pushChannel.ChannelUri.AbsoluteUri);
    }
    catch (Exception ex)
    {
        System.Diagnostics.Debug.WriteLine(ex.ToString());
    }
}
```

I've left out the methods we've already discussed, and you can always refer to the code.

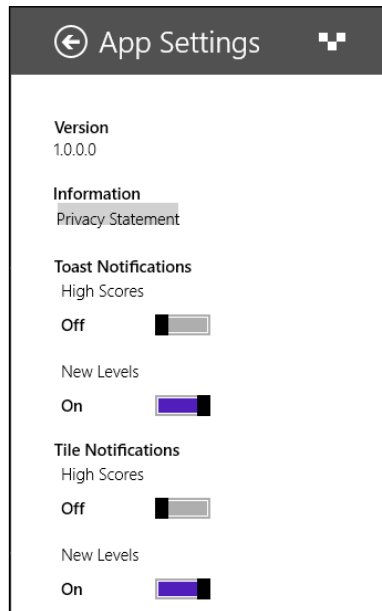
Implementing settings pages

In the Windows Phone game, I've put in a **settings** page (`Settings.xaml`) that has a single `ToggleSwitch` to control whether the push channel is open and registered with the hub or our service. The `ToggleSwitch` is bound to a property in the view model, which exposes the `SettingsHelper.Default.IsPushEnabled` property and calls the `ChannelHelper.Default.SetupChannel` method on change. This is shown in the following screenshot:



There is also **Version** information and a button that launches a web browser with our privacy policy using the `WebBrowserTask` method. Please refer to the code for full view and view model implementations.

Windows Store apps of course have a dedicated settings panel exposed via the Charm bar. Fortunately, Windows 8.1 has a new `SettingsFlyout` control, which makes creating settings flyouts vastly easier than in 8.0. Also, `AppSettingsFlyout.xaml` allows the user to choose categories that they want to be notified about and the type of notification. The toggle switches bind to properties in the `TagHelper` class and calls the `TileTapperPush.UploadChannel;` method on change:



There is also **Version** information and a button which launches a web browser with our privacy policy using the `Launcher.LaunchUriAsync` method. As with the phone app, please refer to the code for full view and view model implementations.

Summary

Well, we've reached the end of the book and covered all the things we need to develop our applications enabled with Windows Azure Mobile Service. By this point, we've probably got some polishing up to do in our code and UI (I know I have), but we can also get our service ready for production by doing the following things:

- Have a look at the logs and make sure there are no errors you need to fix.
- Turn off the automatic database schema function. In the portal's **CONFIGURE**, tab under the **Dynamic Schema** section, disable the **ENABLE DYNAMIC SCHEMA** switch.

- Review the *Rules for choosing permissions* section in *Chapter 3, Securing Data and Protecting the User*, and check if the permissions are correct on all the tables and APIs.
- Pull a copy of the scripts using Git and back them up.
- Check any scheduled tasks are scheduled properly, if required.
- Check your scaling configuration.

Once we're live, we can then use the dashboard to monitor how our services are performing and tune them once the apps are live.

Hopefully, you've enjoyed the book. I've had fun writing it! I've tried to put a lot of stuff into the code samples to help round off the book, so make sure you check these out too and feel free to copy and reuse as much as you can. The source is available at <http://www.packtpub.com>.

Index

A

addToQueue function 73

ANS 61

API scripts

about 50

creating 50, 51

high-score API script 51

HTTP methods 50

app authentication

about 35

credentials, storing 37-39

DataServiceBase class 40, 42

logging in 35, 36

logging out 39

Apple Notifications Service. *See* **ANS**

Apple Push Notification Service (APNs) 15

application key 13

apps

creating, from portal 21-23

audience

targeting, tags used 91-93

B

Backend services 90

badge notifications, WNS scripts

sending 88

BASIC mode 16

C

ChannelHelper class 101

ChannelHelper.Default.SetupChannel

method 103

Configure tab

App settings 16

Cross-origin resource sharing (CORS) 16

Database settings 15

Developer analytics 16

Dynamic schema 16

monitoring 16

Source control 15

Create button 25

Create New Table dialog 26

Cross-origin resource sharing (CORS) 16

CurrentUser property 36

D

Database settings 15

DataService method 65

dequeue method 74

development environment

hardware requirements 19

preparing 19

software, setting up 20

store accounts, requirements 20

Dynamic schema 16

E

Everyone option 32

existing apps, connecting to Windows Azure Mobile Services

Connected Service, adding 24

SDK manual installation, in Visual Studio

2012 Express 25

table, creating 25, 26

table, interacting with 27-29

table model, writing 26

exports method 51

F

Fiddler 20, 43

G

GCM 15

GetAll method 39

Git

- changes, pushing 57
- repository, pulling 56
- repository, updating 57
- scripts, adding manually 57
- working with 56

Git Bash 56

Google Cloud Messaging *See* GCM

H

high-score API script 51-53

HTTP methods

- Insert 45
- Query 45
- Update 45

Hub. *See* Notifications Hub

Hyper-V phone emulators 19

I

Insert method 27

InvokeApiAsync method 53

J

JSON Web Token (JWT) 35

K

key management

- application key 13
- master key 13

L

LeaderBoard table 26

level-insert table script

- example 48

LoginAsync method 36

M

MANAGE KEYS button 14

master key 13, 43, 45

MessageDialog class 99

Microsoft account 7

Microsoft Live ID. *See* Microsoft account

Microsoft Push Notification Service. *See*

MPNS

mobile service

- Configure tab 15
- creating 10-12
- dashboard 14
- features 13
- keys, managing 13
- logs tab 18
- scalability 16, 17

MobileServiceClient class 35, 39

mobile service dashboard

- API tab 15
- autoscale status 14
- Data tab 15
- Identity 15
- mobile service endpoint status 14
- Push 15
- Quick glance section 14
- Scheduler 15
- usage overview 14

MOBILE SERVICE TIER 16

Model View View-Model (MVVM) pattern 27

MPNS 61

MPNS scripts

- creating 88
- tile notifications, sending 78, 89
- toast notifications, sending 77, 89
- Windows Phone apps 77

MVC Web API service 53

N

NetworkAddressChanged event 99

network connection

- checking 98, 99

NetworkInformation.GetInternetConnectionProfile method 98

Node NPM module 58

noscript parameter 45

Notifications Hub

- benefits 81
- calling, from scripts 86
- configuring 82, 83
- disadvantages 82
- flow 81

notifications settings

- managing 100, 102

NPM modules

- about 20
- implementing 58

O

OAuth2 authentication providers

- Facebook 33
- Google 33
- Microsoft 33
- Twitter 33

P

Pay-as-you-go (PAYG) account 8

permission

- configuring 32

permission configuration

- authentication 33
- choosing, rules 32
- OAuth2 authentication providers 33
- Windows Live Connect Single Sign-on, registering 34, 35

Platform property 65

PNS 62

PREVIEW tag 13

privacy policy

- implementing 98

Purchase button 9

Push Notification Service. *See* PNS

push notifications (Windows Store apps)

- URL 97

R

require method 59

REST API 43-45

S

scalability

- capacity 16
- general 16
- SQL Database 17

SCALE-BY METRIC 17

scheduled script

- creating 55, 56

SCHEDULER tab 54

score-insert script

- example 49

score-read script

- example 49

scripts

- about 47
- API scripts 50
- debugging 54
- level-insert table script 48
- score-insert script 49
- score-read script 49
- table scripts 47, 48

sendBadge method 77

sendFlipTile method 78

send method 75

sendNotifications function 72

sendTileSquareText01 method 75

sendToast method 77

sendToastText04 method 75

service scripts

- about 72, 73
- WNS scripts 74

SettingsHelper.Default.IsPushEnabled property 103

settings pages

- implementing 103, 104

Source control 15

STANDARD mode 16

subscription

- basic subscription 8
- free trial 8
- Pay-as-you-go subscription 8
- selecting 7
- Standard subscriptions 8

Surface Pros 20

T

table scripts 47, 48

TagHelper class 91, 100, 104

tags

used, for targeting audience 91, 93

tile notifications, MPNS scripts

sending 89

tile notifications, WNS scripts

sending 87

tiles and badges (Windows Store apps)

URL 97

tiles, Windows Store apps

badges, setting up 66, 67

setting up 66

toast notifications, MPNS scripts

sending 89

toast notifications (Windows Store apps)

URL 97

toast notifications, WNS scripts

sending 87

ToggleSwitch method 103

TRUNCATE button 23

U

UploadChannel method 65

user experience (UX) 95

UX guidelines

URL 96

V

Visual Assets tab 66

Visual Studio 20

W

Windows App

setting up 85

Windows Azure account

creating 9

Windows Azure Fabric Controller 16

Windows Azure Mobile Services

features 31

Windows Notifications Service. *See* **WNS**

Windows Phone 8 apps

setting up 68, 69, 72, 84, 85

Windows Push Notification Services (WNS)
15

Windows Store

App certification, requisites 95, 96

setting up 84, 85

Windows Store apps

about 63

setting up 63, 64

tiles, setting up 66

WNS 61

WNS scripts

badge notifications, sending 77, 88

creating 87

for Store apps 74

multiple tiles, sending 76

tile notifications, sending 75, 87

toast notifications, sending 75, 87



Thank you for buying Learning Windows Azure Mobile Services for Windows 8 and Windows Phone 8

About Packt Publishing

Packt, pronounced 'packed', published its first book "Mastering phpMyAdmin for Effective MySQL Management" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: www.packtpub.com.

About Packt Enterprise

In 2010, Packt launched two new brands, Packt Enterprise and Packt Open Source, in order to continue its focus on specialization. This book is part of the Packt Enterprise brand, home to books published on enterprise software – software created by major vendors, including (but not limited to) IBM, Microsoft and Oracle, often for use in other corporations. Its titles will offer information relevant to a range of users of this software, including administrators, developers, architects, and end users.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

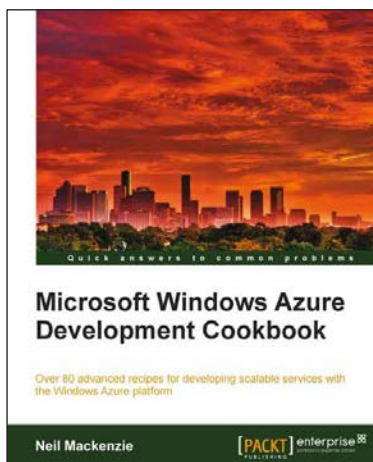


Microsoft Azure: Enterprise Application Development

ISBN: 978-1-84968-098-1 Paperback: 248 pages

Straight talking advice on how to design and build enterprise applications for the cloud

1. Build scalable enterprise applications using Microsoft Azure
2. The perfect fast-paced case study for developers and architects wanting to enhance core business processes
3. Packed with examples to illustrate concepts



Microsoft Windows Azure Development Cookbook

ISBN: 978-1-84968-222-0 Paperback: 392 pages

Over 80 advanced recipes for developing scalable services with the Windows Azure platform

1. Packed with practical, hands-on cookbook recipes for building advanced, scalable cloud-based services on the Windows Azure platform explained in detail to maximize your learning
2. Extensive code samples showing how to use advanced features of Windows Azure blobs, tables, and queues

Please check www.PacktPub.com for information on our titles



Microsoft SQL Azure: Enterprise Application Development

ISBN: 978-1-84968-080-6 Paperback: 420 pages

Build enterprise-ready applications and projects with SQL Azure

1. Develop large scale enterprise applications using Microsoft SQL Azure
2. Understand how to use the various third party programs such as DB Artisan, RedGate, ToadSoft etc developed for SQL Azure
3. Master the exhaustive Data migration and Data Synchronization aspects of SQL Azure



Windows Azure Programming Patterns for Start-ups

ISBN: 978-1-84968-560-3 Paperback: 292 pages

A step-by-step guide to create easy solutions to build your business using Windows Azure services

1. Explore the different features of Windows Azure and its unique concepts
2. Get to know the Windows Azure platform by code snippets and samples by a single start-up scenario throughout the whole book
3. A clean example scenario demonstrates the different Windows Azure features

Please check www.PacktPub.com for information on our titles