

App Clinic Lab Manual

The Field Engineer App

App Clinic Lab Manual

The Field Engineer App

This lab provides an introduction to the main design and functional aspects of a Windows Store-type field service management app. It shows participants how to create a simple scheduling interface using tiles, tied to details pages with associated job and client information. Attendees begin with a new project and go about the task of building the various pieces of the application. The solution uses code snippets to enable the rapid development of the complete application within the scope of a 5-6 hour, instructor-led session.

Windows 8.1 App Clinic

9.17.2013 | Version 1.0

This document is provided “as-is.” Information and views expressed in this document, including URL and other Internet Web site references, may change without notice. Some examples depicted herein are provided for illustration only and are fictitious. No real association or connection is intended or should be inferred. This document does not provide you with any legal rights to any intellectual property in any Microsoft product. You may copy and use this document for your internal, reference purposes.

Copyright © 2013 Microsoft Corporation. All rights reserved.

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

Contents

[Introduction 1](#_Toc367110173)

[Overview 2](#_Toc367110174)

[Welcome to the App Clinic Field Engineer Lab 2](#_Toc367110175)

[Objectives 2](#_Toc367110176)

[Lab Structure 2](#_Toc367110177)

[Estimated Time for Completion 3](#_Toc367110178)

[Getting Started 4](#_Toc367110179)

[System Requirements 5](#_Toc367110180)

[Hardware 5](#_Toc367110181)

[Software 5](#_Toc367110182)

[Solution Overview 6](#_Toc367110183)

[Solution Structure 6](#_Toc367110184)

[Navigation Structure 7](#_Toc367110185)

[Getting to Know the Application 8](#_Toc367110186)

[Branding Considerations 9](#_Toc367110187)

[Understanding Visual Elements 9](#_Toc367110188)

[Image Assets/Logos 9](#_Toc367110189)

[Understanding XAML Styling 10](#_Toc367110190)

[Data Considerations 12](#_Toc367110191)

[Data Model 12](#_Toc367110192)

[Data Schema 14](#_Toc367110193)

[Lab Exercises 16](#_Toc367110194)

[Exercise 1: Local Data Binding 17](#_Toc367110195)

[Task 1: Set Up the Data XML files 17](#_Toc367110196)

[Task 2: Define the Data Source 17](#_Toc367110197)

[Task 3: Fetch and Bind the Data 19](#_Toc367110198)

[Exercise 2: Branding 20](#_Toc367110199)

[Task 1: Configure Visual Assets 20](#_Toc367110200)

[Task 2: Customize Background & Foreground Colors 21](#_Toc367110201)

[Task 3: Customize Text Styles 22](#_Toc367110202)

[Task 4: Customize Data Templates 24](#_Toc367110203)

[Exercise 3: Search 27](#_Toc367110204)

[Task 1: Add Search Declaration 27](#_Toc367110205)

[Task 2: Add Search Results Page 29](#_Toc367110206)

[Task 3: Update Data Source to Return Search Results 30](#_Toc367110207)

[Task 4: Update Binding for Search Results Page 31](#_Toc367110208)

[Task 5: Update Search Results Page Styles 35](#_Toc367110209)

[Exercise 4: Share 38](#_Toc367110210)

[Task 1: Add Handler for Share Charm Invocation 38](#_Toc367110211)

[Task 2: Define the Content Structure to Share 41](#_Toc367110212)

[Exercise 5: App Bar 43](#_Toc367110213)

[Task 1: Add Bottom App Bar 43](#_Toc367110214)

[Task 2: Add Command to App Bar 43](#_Toc367110215)

[Exercise 6: Live Tiles 45](#_Toc367110216)

[Task 1: Choose the Type of Live Tile 45](#_Toc367110217)

[Task 2: Configure the Content for the Live Tile 46](#_Toc367110218)

[Task 3: Enable the Live Tile 46](#_Toc367110219)

[Optional Exercises 48](#_Toc367110220)

[Exercise 1: Semantic Zoom 49](#_Toc367110221)

[Task 1: Add Semantic Zoom Control 49](#_Toc367110222)

[Task 2: Define ZoomedOutView 49](#_Toc367110223)

[Exercise 2: Windowing Modes 52](#_Toc367110224)

[Task 1: Define the Minimum Width for the Application 52](#_Toc367110225)

[Task 2: Define the Narrower View for the Page 52](#_Toc367110226)

[Task 3: Set Up the Visual States 54](#_Toc367110227)

[Task 4: Handle the Window.SizeChanged Event 55](#_Toc367110228)

[Exercise 3: Secondary Tiles 57](#_Toc367110229)

[Task 1: Add the Pin To Start Button 57](#_Toc367110230)

[Task 2: Handle the Button Event 57](#_Toc367110231)

[Task 3: Enable Secondary Tile Pinning 58](#_Toc367110232)

[Task 4: Enable Secondary Tile Unpinning 61](#_Toc367110233)

[Task 5: Enable Entry Through the Secondary Tile 63](#_Toc367110234)

[Conclusion 64](#_Toc367110235)

[Lab Completion 65](#_Toc367110236)

[Achievement Summary 65](#_Toc367110237)

[Appendix 66](#_Toc367110238)

[Table of Figures 67](#_Toc367110239)

[Table of Code Samples 69](#_Toc367110240)

# Introduction

## **Overview**

### Welcome to the App Clinic Field Engineer Lab

One of three available for the **Windows 8.1 App Clinic**[[1]](#footnote-2) workshop, this lab provides an introduction to the main design and functional aspects of a Windows Store-type field service management app. It shows you how to create a simple scheduling interface using tiles, tied to Job Details pages with associated job and client information.

You and your fellow attendees begin with a new project and go about the task of building the various pieces of the application. The solution uses code snippets that will help you rapidly develop the complete application within the scope of a 5-6 hour, instructor-led session.

#### About the labs

The App Clinic labs have a unique structure, in that participants are invited to choose one of three tracks for their development activities. The other two lab tracks, while providing a similar exploration of common Windows Store-style app features, are devoted to retail catalog management and investment banking scenarios. The intent is to give all participants—who, after all, hail from diverse industries and have different business needs—a measure of choice in terms of the kind of app they will each spend the day learning how to build.

One or the other lab track may be more interesting or useful to you because its scenario may be directly or indirectly relevant to the kind of business you are in or the specific business problem you are trying to solve. Your business may need to manage and represent a catalog of goods or services to customers, and so the Contoso Electronics lab track is potentially a good choice for you. Or, perhaps your business is looking for ways to sharpen its field service operations, so the Field Engineer lab track may be the most sensible choice. Even if none of these scenarios exactly fits your circumstances, any one of the labs will give you a solid understanding of several common features of the modern Windows app style and fundamental practices you can take home and immediately turn to the purpose of developing well-designed, innovative apps that help drive business success.

### Objectives

The key objectives for the lab are three:

* Learn the fundamentals of developing a Windows 8.1, Windows Store-style app based on one of three models.
* Using your data and branding, build a working prototype app for your business.
* Gain a clear understanding of next steps and resources for further exploration.

### Lab Structure

After an introduction and overview of Windows 8.1 and demos of the full versions of the apps featured in the App Clinic, the lab will dive into the lab sessions, which will be a combination of hands-on exercises and presentations, as described in Table 1 below.

| Lab Topic | Duration | Type |
| --- | --- | --- |
| Data binding | 60 mins | HOL |
| Custom branding, start screen & logo | 60 mins | HOL |
| Search & search suggestions | 30 mins | HOL |
| Share | 30 mins | HOL |
| App bar | 30 mins | HOL |
| Semantic zoom | 15 mins | PPT |
| Windowing modes | 15 mins | PPT |
| Live tiles | 30 mins | HOL |
| Secondary tiles & notifications | 30 mins | PPT |
| HOL = Hands-on-Lab | PPT = Presentation | | |

Table . Duration and type of topics treated in the lab

At the end of the day, once the exercises are complete, the remaining time—roughly an hour—will be spent on a discussion of best practices and a final wrap-up with next steps.

### Estimated Time for Completion

The full set of lab exercises require approximately 6 hours in an instructor-led session, with appropriate time built in for a mid-session break. On his/her own, an advanced student can conceivably complete the exercises in a much shorter period of time, depending on experience and/or aptitude.

# Getting Started

## System Requirements

This section provides an overview about the hardware and software requirements for this lab. This lab requires the **Windows 8.1 Preview** version, so please make sure that your PC is running this version of the Windows operating system before coming to this lab.

NOTE ▶ There are two ways to install Windows 8.1 Preview: either through the Windows Store or by using an ISO image file (see <http://windows.microsoft.com/en-us/windows-8/preview-download>).

* If you're running Windows 8 in one of the 13 languages for which the preview is available, we strongly recommend that you download the software from the Windows Store.
* If you're running Windows 8 and you want to install Windows 8.1 Preview using the ISO image file, it's important that you know the location of the Windows 8 product key. When you update to the final edition of Windows 8.1, you'll need your Windows 8 product key to activate.
* If you're running Windows XP, Windows Vista, or Windows 7, you can download the ISO image file and use it to create a bootable DVD or USB flash drive.

### Hardware

The following are the system requirements for Windows 8.1 Preview. Note that Windows 8.1 Preview works well on the same hardware that powers Windows 8.

* **Processor:** 1 gigahertz (GHz) or faster
* **RAM:** 1 gigabyte (GB) (32-bit) or 2 GB (64-bit)
* **Free hard disk space:** 16 GB (32-bit) or 20 GB (64-bit)
* **Graphics card:** Microsoft DirectX 9 graphics device with WDDM driver

NOTE ▶ In order to use Windows 8.1 Preview you must sign in to your PC with a Microsoft account. The option to create a local account will be made available with the final release of Windows 8.1.

### Software

To complete the lab exercises, at a minimum you need **Microsoft Visual Studio Express 2013 Preview for Windows** as your development tool. This tool requires Windows 8.1 Preview to be running on your PC; the system requirements are as stated above.

Visual Studio Express 2013 Preview for Windows provides the necessary tools for creating innovative and compelling Windows Store apps. Included are a full-featured code editor, a powerful debugger, a focused profiler, and rich language support that you can use to build apps that you've written in HTML5/JavaScript, C++, C#, or Visual Basic. It also includes a device simulator that you can use to test Windows Store apps on multiple kinds of devices.

NOTE ▶ You can download Microsoft Visual Studio Express 2013 Preview for Windows and find additional information about the software at <http://www.microsoft.com/en-us/download/details.aspx?id=39313>.

## Solution Overview

This section provides an overview of the solution that will be used throughout the lab exercises, covering project structure, application functionality, and navigation within the application. For this lab, you will be using as your template the Field Engineer application, a Windows Store-style field service management app designed for a service technician working for the fictitious Contoso Satellite Network. The application illustrates a typical service call scenario and demonstrates how a Windows 8.1 tablet app can help field engineers better manage their schedules and engage customers in the field.

### Solution Structure

In this lab, you will be using the **AppClinic-FieldEngineer** solution. You will find two sets of folders in the solution, representing the starting and completed states of the lab, respectively labeled “Start” and “Final,” as described below.

* **FieldEngineer-Start:** Represents the initial state of the project to be used as the starting point for lab exercises.
* **FieldEngineer-Final:** Represents the final state of the project to be used as the reference project.



Figure . The structure of the Field Engineer solution

The key folders within the solution are described below in Table 2.

|  |  |
| --- | --- |
| Folder Name | Purpose/ContenTS |
| Assets | Dedicated to all branding logos. The following items are placed inside this folder:   * 4 tile images (150 x 150, 310 x 150, 310 x 310, 70 x 70) * A store logo (50 x 50) * A splash-screen image (620 x 300) * A search logo (30 x 30) |
| Branding | For the custom theme images, custom styles, and custom template XAML files. See the “Branding Considerations” section below for more information. |
| Common | For the common code base, shared across the application. Converters, helpers, and custom types (not data models) are placed inside this folder. |
| Data | Dedicated to offline data elements (XML and artifacts). The following are placed inside this folder:   * Equipments.xml – Equipment list data * Jobs.xml – Jobs list data * EquipmentImages folder – For equipment images |
| DataModel | For all custom data model types. Refer to the “Data Considerations” section below for more information. |
| DataSource | Contains a couple of sealed class definitions that help us to de-serialize offline data stored in the Data folder.  These class files act as data sources for the entire application. The static sync methods defined inside are accessed whenever needed. |
| *<Root folder>* | All screen XAMLs are placed in the root directory. You can also find the Package.appxmanifest and TemporaryKey files here. |

Table . Contents of each of the solution folders

### Navigation Structure

The Field Engineer application uses the hierarchical system of navigation. This pattern, commonly used by Windows Store apps, provides a fast and fluid navigation. It is best for apps with large content collections or many distinct sections of content for a user to explore.

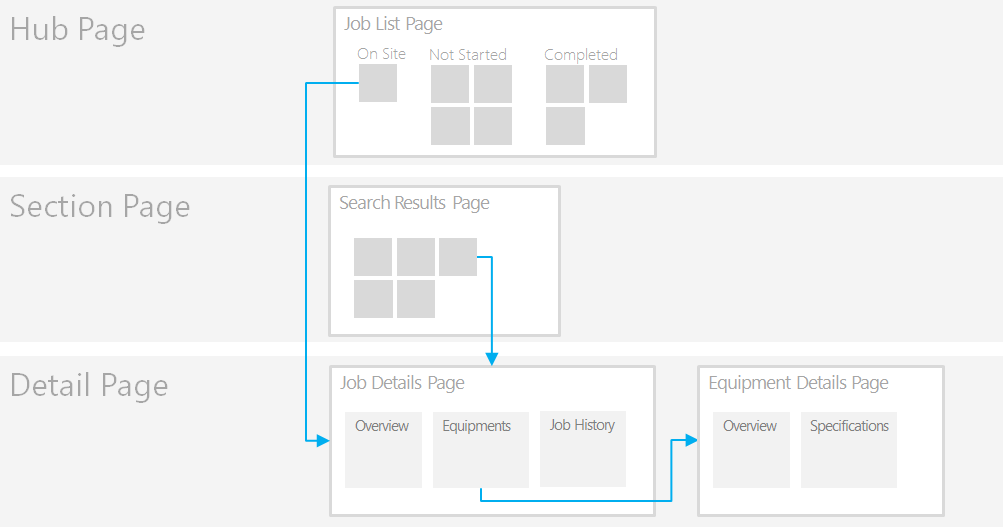


Figure . Simple wireframe showing how the hierarchical navigation system works

NOTE ▶ For further guidance regarding the various navigation patterns for Windows Store apps, see: <http://msdn.microsoft.com/en-us/library/windows/apps/hh761500.aspx#hierarchical_system>

### Getting to Know the Application

Before you start the lab exercise, take a moment to get familiar with the application. Just build and run the **ContosoElectronics-Final** project and closely inspect the application, including its Windows 8.1 platform features.

Figure 3 below shows the welcome page of the application, which lays out the various categories and jobs.

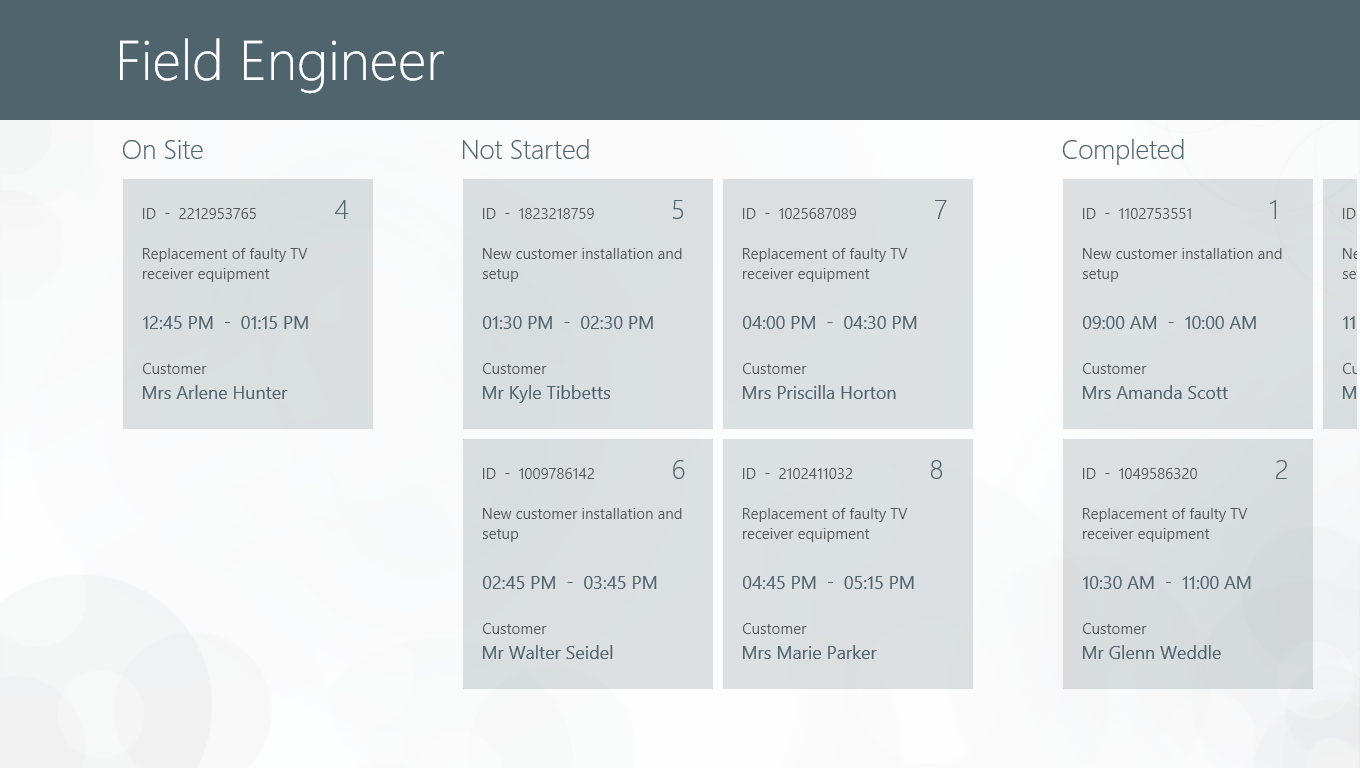


Figure . Welcome page for the Field Engineer app

#### Suggested activities

Take a moment to navigate throughout the application using the navigation structure described above. As you explore, perform the following tasks:

* Observe the live tile and the way it displays job details.
* Try out the various Start screen tile sizes.
* Try Semantic Zoom on the landing page.
* Search for jobs using the Search charm.
* Share a job details page using the Share charm.
* Pin a job on the Start screen from its details page.

## Branding Considerations

A brand is generally defined by the qualities for which a business wants to be known—its identity in the marketplace. When designing your Windows Store apps, you need to ensure that your apps incorporate the essence of your brand. This is both for the sake of overall consistency in the visual design of your organization’s assets—its “brand cohesion”—and the need to differentiate your apps from all the other apps available in the Windows Store, in particular those belonging to your competitors.

### Understanding Visual Elements

The expression of your brand is achieved through a set of visual elements—for example, a distinctive color palette, graphics, layout, and photography style. These elements work together to create a repeatable and recognizable visual system, consistently executed across a variety of media types and digital assets, such as broadcast, print, and Web marketing materials, Web sites, applications, signage, etc. Think of these visual elements as the knobs and dials you manipulate through code to give your Windows Store app a unique look-and-feel.

|  |  |
| --- | --- |
| VISUAL Element | Description |
| Colors | Color is a key attribute for expressing brand. Apply the primary color associated with your brand in ways that tell people that this app comes from your business. |
| Graphics | Use graphics to reinforce your brand by adding character to the presentation of content. Too many graphics, however, can interfere with the flow of your content and can become mere decoration or distraction. |
| Images | Illustrations and photography should also reflect your brand. Reuse the same imagery and style from your company's other media or websites. |
| Grid | The Windows Store app grid system helps unify the visual elements of your app's presentation. The grid aligns your branded app's UI to work with the rest of Windows. |
| Layout | The arrangement of visual elements for all pages needs to be relevant to your brand. You should also strive for consistency across page and content types. |
| Logo | Use your logo to help people quickly identify your app and recognize your brand. |
| Typography | Typefaces are a key part of Windows Store apps. The right choice can be as impactful to your brand as color, logo, or layout, so be thoughtful about the typography you use. |

Table . Visual elements described

### Image Assets/Logos

A Windows Store app includes a basic set of image assets—such as logos or representative icons—for which there are some basic requirements and considerations to bear in mind. The assets specifically required for this lab are described below.

| IMAGE ASSETS | SIZE | DESCRIPTION |
| --- | --- | --- |
| Logo – Small Tile | 70 x 70 | For the **small** square tile image of the app in the Start screen. |
| Logo – Medium Tile | 150 x 150 | For the **medium** square tile image of the app in the Start screen. |
| Logo – Large Tile | 310 x 310 | For the **large** square tile image of the app in the Start screen. |
| Logo – Wide Tile | 310 x 150 | For the **wide** tile image of the app in the Start screen. While not strictly required, without it, a user will be unable to resize the tile to a wide format, nor benefit from the additional detail available in wide-formatted notifications. |
| Small Logo | 30 x 30 | Displayed with your app's display name in search results returned on the Start screen; also used in the list of searchable apps and in the zoomed-out view of the Start screen. |
| Store Logo | 50 x 50 | For the Windows Store when it displays your app's listing in search results and together with the app's description in the listing page. |
| Badge Logo | 24 x 24 | Exclusively used for display next to the badge notification to identify the app on the lock screen; must be monochromatic. |
| Splash Screen Image | 620 x 300 | Shown on your app’s splash screen, which appears for a short interval when the user first opens the app. A mandatory element for which you can customize the background color. |

Table . Image assets required for the lab

NOTE ▶ Bear in mind the following considerations as you design and implement your image assets:

* All the above images assets/logos are defined within the Package.appxmanifest file of the project. The **Visual Assets** section of the **Application UI** tab provides a list of all of these image assets.
* Make sure that the various image assets match the exact size specifications defined in the table above.
* To provide a better experience on computers with different screen resolutions, you should consider including scaled versions of the images in your app. Note that the scaled images should be designed for the image size and not simply resized from the original image.
* When designing the images for the Start screen tiles, consider whether you want to include your app's name in the image or have Windows overlay the name on top of the image (see the [MSDN article](http://msdn.microsoft.com/en-us/library/windows/apps/hh694077.aspx) for further information).

### Understanding XAML Styling

It is important to understand how styles work for XAML applications and what you need to do to customize them to incorporate your own branding. This is where the **ResourceDictionary** class and the **ThemedDictionary** property of that class come into play.

#### The ResourceDictionary class

The **ResourceDictionary** contains resources used by components of the app. The purpose of a **ResourceDictionary** is to enable you to define resources in XAML, and then retrieve them through XAML references made with either the [StaticResource](http://msdn.microsoft.com/en-us/library/windows/apps/hh758287.aspx) markup extension or the [ThemeResource](http://msdn.microsoft.com/en-us/library/windows/apps/dn263118.aspx) markup extension. In this way, you can:

* Reuse resources you've already defined once in XAML.
* Control the complexity of your XAML by following best practices for how to factor XAML-defined elements and any shared value
* Ensure that certain values such as brush colors or pixel measurements are used consistently.

#### The ThemeDictionaries property

**ThemeDictionaries** define a collection of merged resource dictionaries that are specifically keyed and composed to address theme scenarios. Each **resource dictionary** within **ThemeDictionaries** contain resources for a specific theme and must have an **x:Key** attribute representing the name of that theme.

The example below represents three resource dictionaries with three different themes. Note that each resource dictionary has a unique **x:Key** attribute, while the **SolidColorBrush** resource has the same **x:Key** attribute across the three resource dictionaries, but with a different color value specific to each theme.

|  |
| --- |
| XAML |
| * + 1. <ResourceDictionary     2. xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"     3. xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">     4. <!—Theme Dictionaries contains resources that vary across themes -->     5. <ResourceDictionary.ThemeDictionaries>     7. <!-- Contains Default theme resources -->     8. <ResourceDictionary x:Key="Default">     9. <SolidColorBrush x:Key="ApplicationForegroundThemeBrush" Color="#DEFFFFFF" />     10. ...     11. </ResourceDictionary>     12. <!-- Contains High Contrast theme resources -->     13. <ResourceDictionary x:Key="HighContrast">     14. <SolidColorBrush x:Key="ApplicationForegroundThemeBrush" Color="#FF000000" />     15. ...     16. </ResourceDictionary>     17. <!-- Contains Light theme resources -->     18. <ResourceDictionary x:Key="Light">     19. <SolidColorBrush x:Key="ApplicationForegroundThemeBrush" Color="#FFCCCCCC" />     20. ...     21. </ResourceDictionary>     22. </ResourceDictionary.ThemeDictionaries>     23. ...     24. </ResourceDictionary> |

Code Sample . Resource dictionaries and theme dictionaries

NOTE ▶ In Windows 8.1, all the standard XAML resources have been moved out of the project template and are now maintained as part of the platform. All resources including brush colors, pixel measurements, control styles, etc. have been moved to a file called Generic.xaml stored in C:\Program Files (x86)\Windows Kits\8.1\Include\Winrt\Xaml\Design.

## Data Considerations

Without the requisite data, a well-designed app is just an empty—albeit pretty—shell. Indeed, as a practical matter, an app can only be as good as the data it’s intended to showcase. Accordingly, success with this lab depends on understanding the offline data and data model classes for the application and how they are organized.

### Data Model

The data for the app finds its structure in the DataModel folder. Below is the list of available classes.

|  |  |  |
| --- | --- | --- |
| Class Name | Location | Purpose/Notes |
| JobGroup | /DataModel/JobGroup.cs | This class represents the group of jobs stamped with a label. |
| Job | /DataModel/Job.cs | Represents a single job card item.  EquipmentIds is a member that keeps the IDs of equipment required to get the job done. |
| JobHistory | /DataModel/JobHistory.cs | Represents a single activity done on a specific job card item. |
| Customer | /DataModel/Customer.cs | Contains basic details about the customer for the job. |
| Equipment | /DataModel/Equipment.cs | Details the equipment for a job. A job card item may have one or more equipment items attached to it. |
| EquipmentSpecification | /DataModel/EquipmentSpecification.cs | Represents a single specification of a piece of equipment. The Equipment class can contain one or more specifications. |

Table . Data classes in the Field Engineer data model

To help you visualize the relationship of the different classes in the model, a class diagram is provided below.

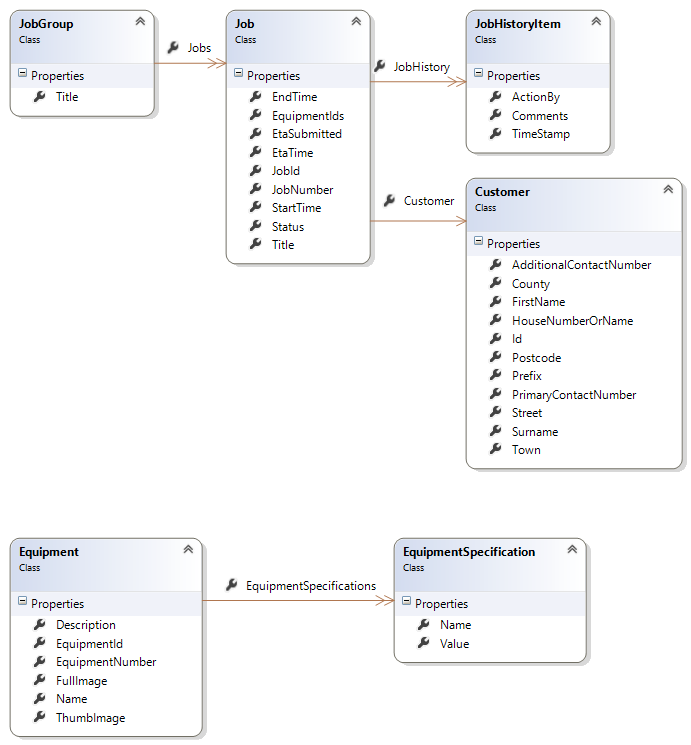


Figure . Class diagram of the Field Engineer data model

### Data Schema

The Field Engineer application uses XML files as the containers for offline data, and can be found in the Data folder. There are two XML files used in this lab, Jobs.xml and Equipments.xml, as described in the table below.

|  |  |
| --- | --- |
| File name | Pupose |
| Jobs.xml | Keeps the list of jobs. Each job has its collection of equipment IDs, collection of activities done so far, and the member for customer information. |
| Equipments.xml | Keeps the list of all equipment. Each piece of equipment has its own list of specifications. |

Table . Schema descriptions

#### Jobs.XML

|  |
| --- |
| XML |
| * + 1. <ArrayOfJob xmlns="http://schemas.datacontract.org/2004/07/FieldEngineer.DataModel" xmlns:i="http://www.w3.org/2001/XMLSchema-instance">     2. <Job>     3. <Customer>     4. <AdditionalContactNumber>09876 987654</AdditionalContactNumber>     5. <County>Berkshire</County>     6. <FirstName>Amanda</FirstName>     7. <HouseNumberOrName>20</HouseNumberOrName>     8. <Id i:nil="true"/>     9. <Postcode>RG6 1WG</Postcode>     10. <Prefix>Mrs</Prefix>     11. <PrimaryContactNumber>01234 123456</PrimaryContactNumber>     12. <Street>Demo Way</Street>     13. <Surname>Scott</Surname>     14. <Town>Bracknell</Town>     15. </Customer>     16. <EndTime>10:00 AM</EndTime>     17. <EquipmentIds xmlns:a="http://schemas.microsoft.com/2003/10/Serialization/Arrays">     18. <a:string>1</a:string>     19. <a:string>3</a:string>     20. <a:string>5</a:string>     21. </EquipmentIds>     22. <EtaSubmitted>True</EtaSubmitted>     23. <EtaTime>08:30 AM - 09:30 AM</EtaTime>     24. <JobHistory>     25. <JobHistoryItem>     26. <ActionBy>Al Pardo</ActionBy>     27. <Comments>Updated job status to On Site</Comments>     28. <TimeStamp>26/08/2013 08:56 AM</TimeStamp>     29. </JobHistoryItem>     30. <JobHistoryItem>     31. <ActionBy>Al Pardo</ActionBy>     32. <Comments>Sent an ETA of around 08:58 AM</Comments>     33. <TimeStamp>26/08/2013 08:22 AM</TimeStamp>     34. </JobHistoryItem>     35. <JobId>1102753551</JobId>     36. <JobNumber>1</JobNumber>     37. <StartTime>09:00 AM</StartTime>     38. <Status>Completed</Status>     39. <Title>New customer installation and setup</Title>     40. </Job>     41. ...     42. </Job>     43. </ArrayOfJob> |

Code Sample . The Jobs.xml schema

#### Equipments.XML

|  |
| --- |
| XML |
| * + 1. <ArrayOfEquipment xmlns="http://schemas.datacontract.org/2004/07/FieldEngineer.DataModel" xmlns:i="http://www.w3.org/2001/XMLSchema-instance">     2. <Equipment>     3. <Description>3D Over HDMI defines input/output protocols for major 3D video formats, paving the way for true 3D gaming and 3D home theater applications. Audio Return Channel of HDMI cable allows an HDMI connected TV with a built-in tuner to send audio data “upstream” to a surround sound system, eliminating the need for a separate audio cable.</Description>     4. <EquipmentId>1</EquipmentId>     5. <EquipmentNumber>HDMI123456</EquipmentNumber>     6. <EquipmentSpecifications>     7. <EquipmentSpecification>     8. <Name>Cable Length</Name>     9. <Value>1.83 Meters</Value>     10. </EquipmentSpecification>     11. <EquipmentSpecification>     12. <Name>Cable Diameter</Name>     13. <Value>5.5mm</Value>     14. </EquipmentSpecification>     15. <EquipmentSpecification>     16. <Name>Transfer speed</Name>     17. <Value>10.2 Gigabits </Value>     18. </EquipmentSpecification>     19. <EquipmentSpecification>     20. <Name>Full HDCP</Name>     21. <Value>Yes</Value>     22. </EquipmentSpecification>     23. <EquipmentSpecification>     24. <Name>XBOX 360</Name>     25. <Value>Yes</Value>     26. </EquipmentSpecification>     27. </EquipmentSpecifications>     28. <FullImage>/Data/EquipmentImages/HDMI\_1.jpg</FullImage>     29. <Name>DUAL HDMI 1.4 CABLE</Name>     30. <ThumbImage>/Data/EquipmentImages/HDMI\_1\_Thumb.jpg</ThumbImage>     31. </Equipment>     32. ...     33. </ArrayOfEquipment> |

Code Sample . The Equipments.xml schema

# Lab Exercises

## Exercise 1: Local Data Binding

In this exercise, you will setup your custom local data for the Windows Store application. This includes copying your XML & image files representing the custom data, setting up the data source, and finaly binding the data to the pages. (Refer to the **Data Considerations**section above before getting started with this exercise.)

This exercise primarily focuses on the equipment schema to illustrate the data binding within the application. There are three tasks in the exercise:

1. Set up the data XML files.
2. Define the data source.
3. Fetch and Bind the data.

Just follow the steps for each task and use the code samples provided, wherever applicable.

### Task 1: Set Up the Data XML files

1. As a first step, review the existing folders & files associated with the data. Refer to the figure below that illustrates the folder structure with all the files related to data binding.

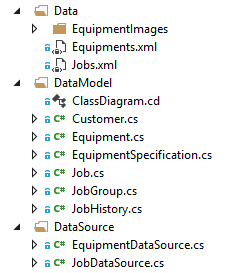


Figure . Folders and files related to data binding

1. Review your custom-updated data XML files as well images. Make sure to update the data model classes within the DataModel folder if there are any changes to the XML schema.
2. Replace the Equipments.xml file within the Data folder with your updated XML file. In case you don’t have any updated file, then use the sample Equipements.xml file provided with the project.

### Task 2: Define the Data Source

1. Open the class file EquipmentDataSource.cs within the DataSource folder and review the existing code.
2. Add a private method **ReadXmlDataFromLocalStorage** within this class to fetch the XML data from the package. Refer to the code sample below.

|  |
| --- |
| C# |
| * + 1. private async Task ReadXmlDataFromLocalStorageAsync()     2. {     3. // Return if data is already loaded     4. if (\_equipmentDataSource.AllEquipments != null)     5. return;     6. try     7. {     8. //Get the folder within the package     9. var dataFolder = await Package.Current.InstalledLocation.GetFolderAsync("Data");     10. //Get the file within the folder     11. StorageFile sessionFile = await dataFolder.GetFileAsync("Equipments.XML");     12. //Read the file     13. using (IRandomAccessStreamWithContentType sessionInputStream = await sessionFile.OpenReadAsync())     14. {     15. var sessionSerializer = new DataContractSerializer(typeof(List<Equipment>));     16. var restoredData = sessionSerializer.ReadObject(sessionInputStream.AsStreamForRead());     17. \_allEquipments = (List<Equipment>)restoredData;     18. }     19. }     20. catch (Exception)     21. {     22. \_allEquipments = null;     23. }     24. } |

Code Sample . Add private method to fetch XML data

1. Add a public method **GetListAsync** within the same class to fetch the list of equipments based upon the equipment Ids. This equipment list is displayed within the Job Details page.

|  |
| --- |
| C# |
| * + 1. /// <summary>     2. /// This method gets all the Equipment using the equipment ids supplied.     3. /// </summary>     4. /// <returns>List of Equipment objects</returns>     5. public static async Task<List<Equipment>> GetListAsync(List<string> equipmentIds)     6. {     7. await \_equipmentDataSource.ReadXmlDataFromLocalStorageAsync();     8. var matches = \_equipmentDataSource.AllEquipments.Where((item) => equipmentIds.Contains(item.EquipmentId)).ToList();     9. return matches;     10. } |

Code Sample . Add public method for fetching equipment list for a job

1. Add another public method **GetDetailsAsync** to fetch equipment details based on the equipment ID. These details are displayed within the equipment details page.

|  |
| --- |
| C# |
| * + 1. /// <summary>     2. /// This method returns the complete details about a Equipment using the Equipment ID.     3. /// </summary>     4. /// <param name="EquipmentId">The Equipment id.</param>     5. /// <returns>Equipment object</returns>     6. public static async Task<Equipment> GetDetailsAsync(string EquipmentId)     7. {     8. await \_equipmentDataSource.ReadXmlDataFromLocalStorageAsync();     9. var matches = \_equipmentDataSource.AllEquipments.Where((item) => item.EquipmentId.Equals(EquipmentId));     10. if (matches != null && matches.Count() == 1) return matches.First();     11. return null;     12. } |

Code Sample . Add public method to fetch equipment details

### Task 3: Fetch and Bind the Data

The next task is to fetch the equipment data from the equipment datasource using the methods we added within the previous task. After fetching the data, we will bind that that data to the various pages within the application.

1. Open the JobDetailsPage.xaml.cs class file and add the following lines of code within the **LoadState** method. Run the application after making this change to verify that the equipment data from your updated XML file is being reflected within the job details page of the application.

|  |
| --- |
| C# |
| * + 1. this.DefaultViewModel["JobStockItems"] = await EquipmentDataSource.GetListAsync(job.EquipmentIds); |

Code Sample . Bind the equipment list

1. Open the EquipmentDetails.xaml.cs class file and add the following lines of code within the **LoadState** method. Run the application after making this change to verify that the equipement data from your updated XML file is being reflected within the equipment details page of the application.

|  |
| --- |
| C# |
| * + 1. // Assign the various bindable items to DefaultViewModel     2. var selectedEquipment = await EquipmentDataSource.GetDetailsAsync( e.NavigationParameter.ToString());     3. this.defaultViewModel["EquipmentDetail"] = selectedEquipment;     4. this.defaultViewModel["EquipmentName"] = selectedEquipment.Name; |

Code Sample . Bind the equipment details

## Exercise 2: Branding

In this exercise, you will customize the branding for the Windows Store application. Branding includes configuring the logos for the various tile sizes, the search and store logos, the background and foreground colors for the application, customizing the text styles and the data templates, etc. (Refer to the *Branding Considerations* section above before getting started with this exercise.)

There are four tasks in this exercise:

1. Configure visual assets (tiles & logos).
2. Customize background and foreground colors.
3. Customize text styles.
4. Customize data templates.

### Task 1: Configure Visual Assets

First, we’ll configure the visual assets for the application, including the tile images and logos, badge logo, splash screen, etc.

1. Open the Assets folder and review the various images/logos within that folder. Replace any/all of these files with your custom images/logos.

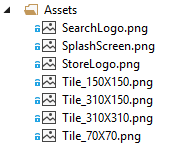


Figure . Contents of the Assets folder

* Make sure that your custom images/logos are as per the size specifications of the visual assets. Refer to Table 4 above for more information.

NOTE ▶ For additional guidance around size specifications, refer to the [MSDN article](http://msdn.microsoft.com/en-us/library/windows/apps/hh846296.aspx#storelogo).

* Make sure you keep the file names the same when your override the files with your custom images/logos. Should the file names be different, you will need to upate the file names within the application manifest file, as explained in the next step.

1. Open the Package.appxmanifest file in the folder root and refer to the **Application UI** tab as shown in the figure below. Make sure that each of the assets within the list points to the correct file name.

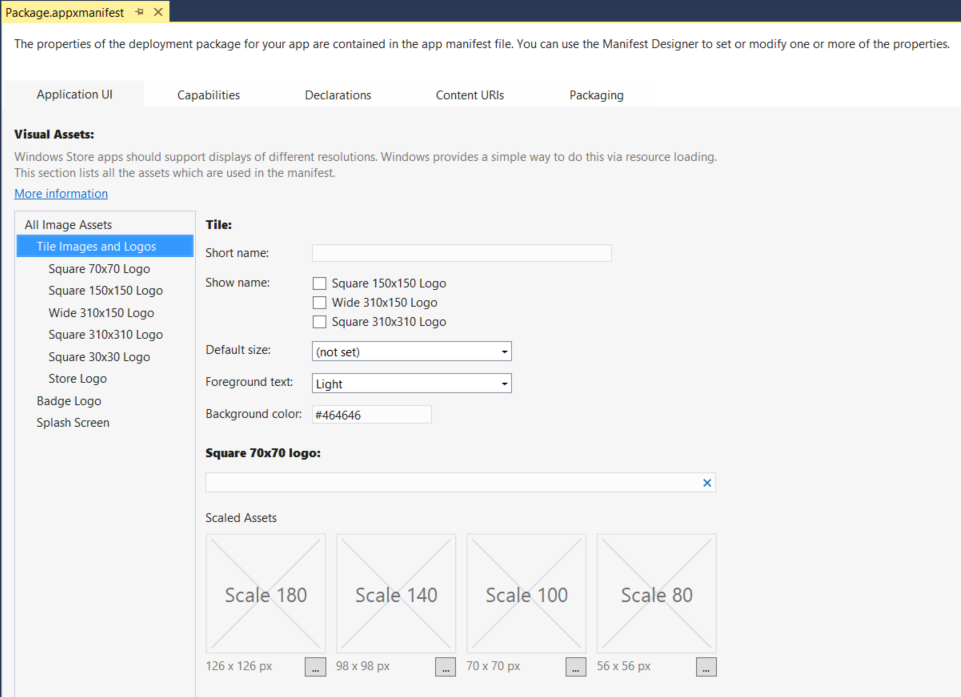


Figure . Configure image asset sizes

### Task 2: Customize Background & Foreground Colors

The next thing to do is to customize the primary and secondary colors for the entire application. This includes setting the background and foreground colors for various UI elements. In this task, we will override some of the key brush colors to reflect your custom branding.

Windows 8.1 provides standard brush colors as XAML resources that defines the primary and secondary colors for an application. These XAML resources are maintained as part of the platform and are stored in a file called **Generic.xaml** located in C:\Program Files (x86)\Windows Kits\8.1\Include\Winrt\Xaml\Design.

1. Open Generic.xaml and take a quick look at some of the standard brush colors defined within the theme dictionaries. Note that there are more than 240 brush colors that are defined for each theme.
2. Open the CustomStyles.xaml file within the Branding folder and locate the section shown below. This section represents a brief description about each color brush. Replace these colors with those that match your branding.

|  |
| --- |
| XAML |
| * + 1. <!--     2. ========================================================================     3. OVERRIDE STANDARD SYSTEM COLOR BRUSH     4. The below are some of the key system defined color brush resources that     5. can be overriden to reflect your branding colors.     7. > AppBarBackgroundThemeBrush: Background Color for the App Bar     8. > AppBarBorderThemeBrush: Border Color for the App Bar     9. > AppBarItemForegroundThemeBrush: Foreground Color for App Bar Items     10. > ApplicationForegroundThemeBrush: Primary Foreground Color     11. > ApplicationHeaderForegroundThemeBrush: Foreground Color for Header Text     12. > ApplicationSecondaryForegroundThemeBrush: Secondary Foreground Color     13. > ApplicationPageBackgroundThemeBrush: Page Background Color     14. > BackButtonBackgroundThemeBrush: Background Color for Back Button     15. > BackButtonForegroundThemeBrush: Foreground Color for Back Button     16. ========================================================================     17. -->     18. <SolidColorBrush x:Key="AppBarBackgroundThemeBrush" Color="#994f646d" />     19. <SolidColorBrush x:Key="AppBarBorderThemeBrush" Color="#FF000000" />     20. <SolidColorBrush x:Key="AppBarItemForegroundThemeBrush" Color="#FFFFFFFF" />     22. <SolidColorBrush x:Key="ApplicationForegroundThemeBrush" Color="#4f646d" />     23. <SolidColorBrush x:Key="ApplicationHeaderForegroundThemeBrush" Color="#FFFFFFFF" />     24. <SolidColorBrush x:Key="ApplicationSecondaryForegroundThemeBrush" Color="#99000000" />     25. <SolidColorBrush x:Key="ApplicationPageBackgroundThemeBrush" Color="#FFFFFFFF" />     26. <SolidColorBrush x:Key="BackButtonBackgroundThemeBrush" Color="Transparent" />     27. <SolidColorBrush x:Key="BackButtonForegroundThemeBrush" Color="#FFFFFFFF" /> |

Code Sample . Overriding default SolidColorBrush resources

1. Open the CustomStyles.xaml file within the Branding folder and locate the section shown below. This section presents a brief description about each custom color and image brush. These color brushes are not system colors but are added for some specific purpose like the background of tiles, page background image, etc. You can replace these colors with those that match your branding.

|  |
| --- |
| XAML |
| * + 1. <!--     2. ========================================================================     3. DEFINE NEW CUSTOM STYLES:     5. PageBackgroundImage: Background Image for all the pages     6. PageHeaderBackgroundBrush: Background Color for the Page Title Header     7. TileBackgroundBrush: Background Color for active Tiles (clickable)     8. InactiveTileBackgroundBrush: Background Color for in-active Tiles     9. ========================================================================     10. -->     11. <ImageBrush x:Key="PageBackgroundImageBrush" ImageSource="/Branding/Background.png" />     12. <SolidColorBrush x:Key="PageHeaderBackgroundBrush" Color="#4f646d" />     13. <SolidColorBrush x:Key="TileBackgroundBrush" Color="#304f646d" />     14. <SolidColorBrush x:Key="InactiveTileBackgroundBrush" Color="#124f646d" /> |

Code Sample . Adding new SolidColorBrush and ImageBrush resources

### Task 3: Customize Text Styles

We’ll now customize the text styles for the entire application, including the font size, font weight, and the foreground colors for the various types of text that appear in the app.

Windows 8.1 provides standard text block styles as XAML resources that are maintained as part of the platform in a file called **Generic.xaml** located in C:\Program Files (x86)\Windows Kits\8.1\Include\Winrt\Xaml\Design.

1. Open **Generic.xaml** and take a quick look at the various text block styles found near the end of this file. Note that there are six standard text block styles shown below that are defined as part of the platform.

|  |
| --- |
| XAML |
| * + 1. <Style x:Key="HeaderTextBlockStyle" TargetType="TextBlock" BasedOn="{StaticResource BaseTextBlockStyle}">     2. <Setter Property="FontSize" Value="56"/>     3. <Setter Property="FontWeight" Value="Light"/>     4. <Setter Property="LineHeight" Value="40"/>     5. </Style>     6. <Style x:Key="SubheaderTextBlockStyle" TargetType="TextBlock" BasedOn="{StaticResource BaseTextBlockStyle}">     7. <Setter Property="FontSize" Value="26.667"/>     8. <Setter Property="FontWeight" Value="Light"/>     9. <Setter Property="LineHeight" Value="30"/>     10. </Style>     11. <Style x:Key="TitleTextBlockStyle" TargetType="TextBlock" BasedOn="{StaticResource BaseTextBlockStyle}">     12. <Setter Property="FontWeight" Value="SemiBold"/>     13. </Style>     14. <Style x:Key="SubtitleTextBlockStyle" TargetType="TextBlock" BasedOn="{StaticResource BaseTextBlockStyle}">     15. <Setter Property="FontWeight" Value="Normal"/>     16. </Style>     17. <Style x:Key="BodyTextBlockStyle" TargetType="TextBlock" BasedOn="{StaticResource BaseTextBlockStyle}">     18. <Setter Property="FontWeight" Value="SemiLight"/>     19. </Style>     20. <Style x:Key="CaptionTextBlockStyle" TargetType="TextBlock" BasedOn="{StaticResource BaseTextBlockStyle}">     21. <Setter Property="FontSize" Value="12"/>     22. <Setter Property="FontWeight" Value="Normal"/>     23. </Style> |

Code Sample . Standard framework TextBlock styles

1. Next, open CustomStyles.xaml in the Branding folder and locate the section shown below. This section presents a brief description of each text block style. Customize these styles to match your branding.

|  |
| --- |
| XAML |
| * + 1. <!--     2. ========================================================================     3. OVERRIDE STANDARD SYSTEM TEXT BLOCK STYLES     4. The below are the key system defined text bloxk styles that can be     5. overriden to reflect your branding colors.     7. > HeaderTextBlockStyle: Style for the Page Header Text     8. > SubheaderTextBlockStyle: Style for the Group/Section Header Text     9. > TitleTextBlockStyle: Style for the Title text within a section     10. > SubtitleTextBlockStyle: Style for the sub title text within a section     12. > BodyTextBlockStyle: Style for the body text     13. > CaptionTextBlockStyle: Style for the caption text     14. ========================================================================     15. -->     16. <Style x:Key="HeaderTextBlockStyle" TargetType="TextBlock" BasedOn="{StaticResource BaseTextBlockStyle}">     17. <Setter Property="FontSize" Value="56"/>     18. <Setter Property="FontWeight" Value="Light"/>     19. <Setter Property="LineHeight" Value="40"/>     20. <Setter Property="Foreground" Value="{StaticResource ApplicationHeaderForegroundThemeBrush}"/>     21. </Style>     22. ... |

Code Sample . Overriding framework TextBlock styles

### Task 4: Customize Data Templates

Lastly for this exercise, we need to customize the structure, layout, and overall design of the data templates, of which there are three in this lab:

* Job item template
* Stock item template
* Job history item template

Let’s take these in order.

1. **The job item template.** This data template provides for the structure and layout for job items, including the data fields shown in the figure below. This data template is used to show the job items on the job list and search results pages.

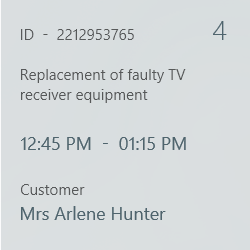


Figure . Data template for job items

Customize the style elements for this template—alignments, font sizes, etc.—in accordance with your brand guidelines and/or the overall design you wish your app to have.

|  |
| --- |
| XAML |
| * + 1. <DataTemplate x:Key="JobItemTemplate">     2. <Grid HorizontalAlignment="Left" Width="250" Height="250" Background="{StaticResource TileBackgroundBrush}">     3. <Grid.RowDefinitions>     4. <RowDefinition Height="40"/>     5. <RowDefinition Height="80"/>     6. <RowDefinition Height="30"/>     7. <RowDefinition Height="\*"/>     8. </Grid.RowDefinitions>     9. <StackPanel Grid.Row="0" Margin="20,20,20,0" Orientation="Horizontal">     10. <TextBlock Style="{StaticResource SubtitleTextBlockStyle}" Margin="0,0,80,0">     11. <Run Text="ID"/>     12. <Run Text=" - "/>     13. <Run Text="{Binding JobId}" />     14. </TextBlock>   <TextBlock Text="{Binding JobNumber}" Style="{StaticResource SubheaderTextBlockStyle}" HorizontalAlignment="Right" />   * + 1. </StackPanel>     2. <StackPanel Grid.Row="1" Margin="20,10,10,10" VerticalAlignment="Center">     3. <TextBlock Style="{StaticResource SubtitleTextBlockStyle}" Text="{Binding Title}" />     4. </StackPanel>     5. <StackPanel Grid.Row="2" Margin="20,10,5,0">     6. <TextBlock Style="{StaticResource TitleTextBlockStyle}" TextWrapping="Wrap">     7. <Run Text="{Binding StartTime}" />     8. <Run Text=" - "/>     9. <Run Text="{Binding EndTime}" />     10. </TextBlock>     11. </StackPanel>     12. <StackPanel Grid.Row="3" Margin="20,0,0,30" VerticalAlignment="Bottom">     13. <TextBlock Style="{StaticResource SubtitleTextBlockStyle}" TextWrapping="Wrap" >     14. <Run Text="Customer"/>     15. </TextBlock>     16. <TextBlock Style="{StaticResource TitleTextBlockStyle}" TextWrapping="Wrap"     17. Margin="0,5,0,0" >     18. <Run Text="{Binding Customer.Prefix}" />     19. <Run Text="{Binding Customer.FirstName}" />     20. <Run Text="{Binding Customer.Surname}" />     21. </TextBlock>     22. </StackPanel>     23. </Grid>     24. </DataTemplate> |

Code Sample . Customize styles in the job item data template

1. **The stock item template.** The stock item data template establishes the structure and layout for equipment items, including the data fields as shown in the figure immediately below. The equipment items are shown on the job details page in the Stock Items section.

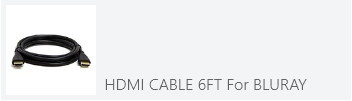


Figure . Data template for stock items

Just as above, customize the style elements for this template—alignments, font sizes, etc.—in accordance with your brand guidelines and/or the overall design you wish your app to have.

|  |
| --- |
| XAML |
| * + 1. <DataTemplate x:Key="StockItemTemplate">     2. <Grid Width="350" Height="100" Background="{StaticResource TileBackgroundBrush}">     3. <StackPanel VerticalAlignment="Center" HorizontalAlignment="Left" Margin="5,0,0,0" Orientation="Horizontal">     4. <Image Source="{Binding ThumbImage}" Height="90" Width="90" Stretch="Fill"/>     5. <TextBlock Text="{Binding Name}" Style="{StaticResource SubtitleTextBlockStyle}" VerticalAlignment="Bottom" TextWrapping="Wrap" Margin="10,0,0,5" Width="250"/>     6. </StackPanel>     7. </Grid>     8. </DataTemplate> |

Code Sample . Customize styles in the stock item data template

1. **The job history item template.** This data template structures and provides the layout for job history items, including the data fields shown in the figure immediately below. The job history items are shown in the Job History section of the job details page.

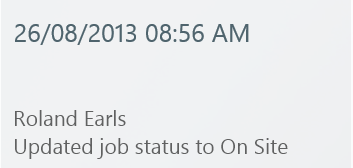


Figure . Data template for job history items

Finally, just as before, customize the style elements for this template—alignments, font sizes, etc.—in accordance with your brand guidelines and/or the overall design you wish your app to have.

|  |
| --- |
| XAML |
| * + 1. <DataTemplate x:Key="HistoryLogItemTemplate">     2. <Grid Width="250" Height="120" Background="{StaticResource InactiveTileBackgroundBrush}">     3. <Grid.RowDefinitions>     4. <RowDefinition Height="\*"/>     5. <RowDefinition Height="2\*"/>     6. </Grid.RowDefinitions>     7. <TextBlock Style="{StaticResource TitleTextBlockStyle}" Text="{Binding TimeStamp}" Margin="10,10,0,0" VerticalAlignment="Top" HorizontalAlignment="Left"/>     8. <StackPanel Grid.Row="1" Margin="10,0,10,10" VerticalAlignment="Bottom" HorizontalAlignment="Left">     9. <TextBlock Text="{Binding ActionBy}" Style="{StaticResource SubtitleTextBlockStyle}"/>     10. <TextBlock TextWrapping="Wrap" Style="{StaticResource SubtitleTextBlockStyle}">     11. <Run Text="{Binding Comments}"/>     12. </TextBlock>     13. </StackPanel>     14. </Grid>     15. </DataTemplate> |

Code Sample . Customize styles in the job history item template

## Exercise 3: Search

Windows Store apps expose search consistently. Swiping from the right edge of the screen or pressing the Windows+C key combination invokes the charm bar. The Search charm can be either context-sensitive or global. When you invoke it from the desktop or the Start screen, you’ll access the system-wide search functionality. Note that the UI and actions required to invoke search are the same for all the apps.

NOTE ▶ This lab uses the Windows 8-style search, as described in this exercise. Windows 8.1 exposes search differently, privileging the implementation of an in-app search box rather than an invocation of search through the charm bar. Regardless, the code for implementing either is simple and consistent, and you should have little difficulty in transitioning from one style to the other.

In this exercise, you will learn how to implement the Search contract for the Field Engineer application. You will configure the application to allow its users to perform a free-text search for jobs based on job ID, job name, customer name, job status, etc.

There are five tasks in this exercise:

1. Add search declaration.
2. Add search results page.
3. Update data source to return search results.
4. Update binding for search results page.
5. Update styling for search results page.

### Task 1: Add Search Declaration

Before we start this task, build and run the application and then evoke the Search charm. You will notice that scope selector of the Search charm does not show the Field Engineer application yet, as shown in Figure 11 below.

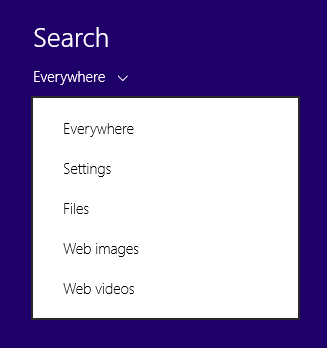


Figure . Search charm Scope Selector

The first task, then, is to add the search declaration so that the Field Engineer starts showing up within the scope selector of the Search charm. Follow the steps below to add the search declaration.

1. Open the Package.appxmanifest file from the Solution explorer and then select the **Declarations** tab.
2. Expand the **Available Declarations** drop-down list box and click **Search**.
3. Click **Add** and then save and close the file. Refer to Figure 12 below, showing the search declaration added to the supported declarations for the app.

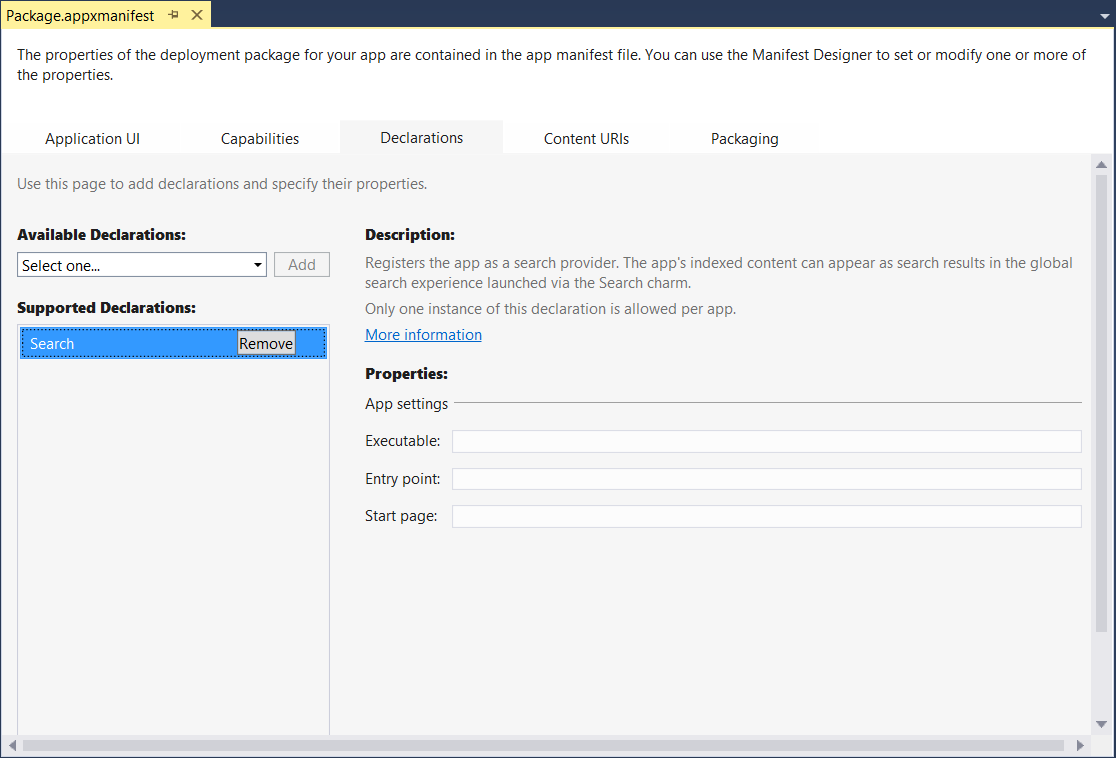


Figure . Search declaration

1. At this point, build and run the application again and then evoke the Search charm. You will notice that the scope selector of the Search charm now shows the Field Engineer app.

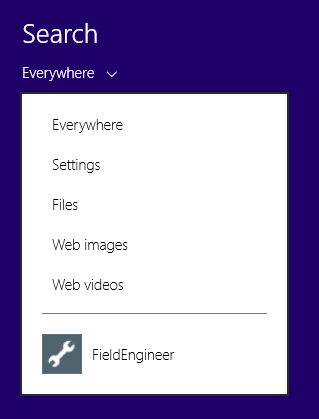


Figure . Search charm scope selector with app now showing

1. Select **Field Engineer** in the scope selector of the Search charm and try to perform a search using any keyword. You will notice that nothing happens. This is because we have yet to create a search results page.

### Task 2: Add Search Results Page

The next task is to add a search results page that will show the search results for a given search query. The search results will comprise a list of jobs matching the search criteria. Note that Visual Studio 2013 provides a default template for the search results page. Follow the steps below to add the search results page.

1. In the Solution Explorer, right-click the project **FieldEngineer-Start** and then select **Add New Item**. You should now see the Add New Item dialog box, as shown below.
2. In the left pane, select **Windows Store**, which appears in the **Visual C#** list. This will show only the Windows Store-specific templates.
3. Scroll down the list of templates and select **Search Results Page**. Change the name of the file to “SearchResultsPage.xaml” in the **Name** box and click **Add**.



Figure . Add search results page

The next step is to integrate the search results page within the application so that whenever the user conducts a search, the application redirects to this page.

1. Open the application code-behind file App.xaml.cs and override the **OnSearchActivated** method. This method is invoked whenever the application is activated to perform a search.
2. Navigate to **Search Results Page** within this method. Refer to the sample code below. Just copy-and-paste the code below within the App.xaml.cs file.

|  |
| --- |
| C# |
| * + 1. /// <summary>     2. /// Invoked when the application is activated to display search results.     3. /// </summary>     4. /// <param name="args">Details about the activation request.</param>     5. protected override void OnSearchActivated(SearchActivatedEventArgs args)     6. {     7. //Get the frame i.e. the content of the current window     8. Frame frame = Window.Current.Content as Frame;     9. //Navigate to the Search Results Page     10. frame.Navigate(typeof(SearchResultsPage), args.QueryText);     11. } |

Code Sample . Handler method activated on search

1. Now, build and run the application again and evoke the Search charm. Select the app **Field Engineer** using the scope selector of the Search charm and try to perform a search using any keyword. You will notice that you are being redirected to the search results page, as shown below in Figure 15.



Figure . Search results page (default view)

### Task 3: Update Data Source to Return Search Results

This task involves updating the existing data source to fetch the search results based upon a search keyword. Here, we will implement the search functionality only for products, so we will need to update the existing data source for products.

1. Open the JobDataSource.cs class file in the DataSource folder.
2. Add a new method **SearchJobsAsync** within this class. This method will accept the search keyword string as the input parameter and will perform the search within the list of jobs based on job ID, job name, customer name, and job status. Finally, this method will return a filtered list of jobs matching the search criteria.
3. Refer to the sample code below, and copy-and-paste it into the JobDataSource.cs file.

|  |
| --- |
| C# |
| * + 1. /// <summary>     2. /// This method searches for jobs based upon the search text. The search text     3. /// can be a part of the Job ID, Job Title, Job Status or even Customer Name.     4. /// </summary>     5. /// <param name="searchText">The search text.</param>     6. /// <returns>Search results</returns>     7. public static async Task<List<Job>> SearchJobsAsync(string searchText)     8. {     9. await \_jobDataSource.ReadXmlDataFromLocalStorageAsync();     10. var filteredList =     11. \_jobDataSource.AllJobs.Where(     12. item =>     13. item.JobId.Contains(searchText) ||     14. item.Title.ToUpper().Contains(searchText.ToUpper()) ||     15. item.Status.ToUpper().Contains(searchText.ToUpper()) ||     16. item.Customer.Surname.ToUpper().Contains(searchText.ToUpper()) ||     17. item.Customer.FirstName.ToUpper().Contains(searchText.ToUpper()));     18. return filteredList.ToList();     19. } |

Code Sample . Search jobs data source method

### Task 4: Update Binding for Search Results Page

The next task is to bind the search results page with the search results from the data source. In this task, we will fetch the search results using the data source method we just added above, and then bind the results data to the page.

1. Open the SearchResultsPage.xaml file and take a close look at the structure of the page. You will notice use of the **filtersItemsControl** and the **resultsGridView** properties. **ItemsControl** is used to show the search filters while **GridView** is used to display the search results.

NOTE ▶ The default page template for SearchResultsPage.xaml already has significant code for XAML and for the code-behind class file. You will notice that the <Page.Resources> within the XAML defines two instances of **CollectionViewSource** (To read more about the **CollectionViewSource** class, refer to the [MSDN article](http://msdn.microsoft.com/en-us/library/windows/apps/windows.ui.xaml.data.collectionviewsource.aspx)). These instances are bound to **Results** and **Filters** properties of the **DefaultViewModel** for the page. You will also notice significant code in the code-behind class file regarding setting the filters and handling the filter-check event.

1. Open the code-behind file SearchResultsPage.xaml.cs and take a close look at the **navigationHelper\_LoadState** method. This method is responsible for fetching the search results and for setting up the necessary search filters.
2. Add the following private variables in the class; these will be used throughout the class.

|  |
| --- |
| C# |
| * + 1. private List<Job> \_searchResults;     2. private string \_queryText; |

Code Sample . Add private class members

1. Update the **navigationHelper\_LoadState** method and add the code to fetch the search results from the products data source as well as the code to set up the filters. Refer to the code sample below.

NOTE ▶ Since the method for searching jobs is asynchronous within the data source, you will need to use **await** while calling the **SearchJobsAsync** method from the **navigationHelper\_LoadState** method within the page. To use **await**, you must make **navigationHelper\_LoadState** asynchronous.

|  |
| --- |
| C# |
| * + 1. private async void navigationHelper\_LoadState(object sender, LoadStateEventArgs e)     2. {     3. \_queryText = e.NavigationParameter as String;     4. // Fetch the serach results from JobsDataSource     5. var searchResults = await JobDataSource.SearchJobsBySearchTextAsync(\_queryText);     6. // Add the counts for various job status     7. int countOnSite = searchResults.Count(item => item.Status == "On Site");     8. int countNotStarted = searchResults.Count(item => item.Status == "Not Started");     9. int countCompleted = searchResults.Count(item => item.Status == "Completed");     10. // Add the filters based upon the job status     11. var filterList = new List<Filter>();     12. filterList.Add(new Filter("All", searchResults.Count, true));     13. if (countOnSite > 0) filterList.Add(new Filter("On Site", countOnSite, false));     14. if (countNotStarted > 0) filterList.Add(new Filter("Not Started", countNotStarted, false));     15. if (countCompleted > 0) filterList.Add(new Filter("Completed", countCompleted, false));     16. // Communicate results through the view model     17. this.DefaultViewModel["QueryText"] = '\u201c' + \_queryText + '\u201d';     18. this.DefaultViewModel["Filters"] = filterList;     19. this.DefaultViewModel["ShowFilters"] = filterList.Count > 0;     20. this.DefaultViewModel["Results"] = searchResults;     21. } |

Code Sample . Load state method for search results page

1. Add the following namespace in the SearchResultsPage.xaml.cs file.

|  |
| --- |
| C# |
| * + 1. using FieldEngineer.DataModel;     2. using FieldEngineer.DataSource; |

Code Sample . Add required namespace

1. Run the application and observe the results now. You will notice that no job data items are displayed as the related data template is not applied yet.

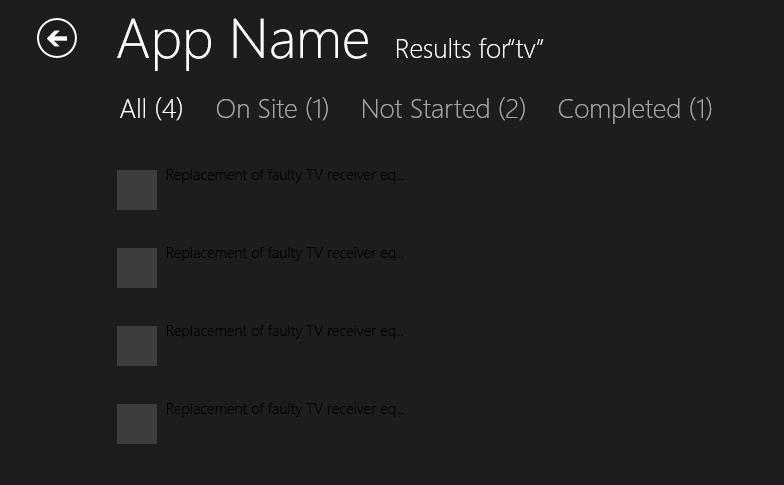


Figure . Search results page

1. The next step is to apply the correct data template to the search results. Open the file SearchResultsPage.xaml and replace the **ItemTemplate** assignment with the following line of code.

|  |
| --- |
| XAML |
| * + 1. ItemTemplate="{StaticResource JobItemTemplate}" |

Code Sample . Assign the correct template

1. Change the height property of the **ItemContainerStyle** to 250 instead of the default 70 so that the template fits correctly.

|  |
| --- |
| XAML |
| * + 1. <GridView.ItemContainerStyle>     2. <Style TargetType="Control">     3. <Setter Property="Height" Value="250"/>     4. <Setter Property="Margin" Value="0,0,38,8"/>     5. </Style>     6. </GridView.ItemContainerStyle> |

Code Sample . Change the item container height

1. Build and run the application. You will notice that the job items are now visible within the search results page as shown in the figure below.

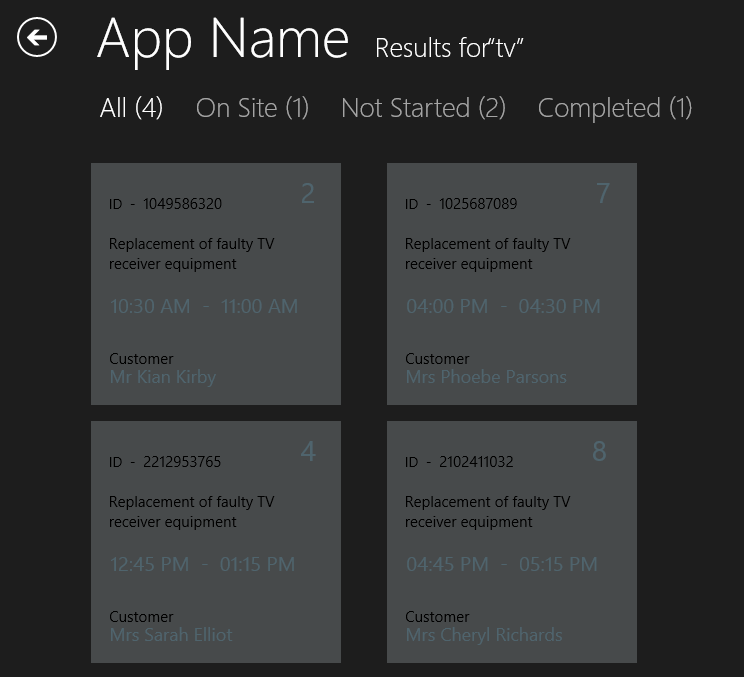


Figure . Search results page with the proper template

1. At this point, the filters still won’t work (try clicking one of them). To handle the filter change event, go to the SearchResultsPage.xaml.cs file and find the **Filter\_Checked** method. Change the method by adding **Async** to the method signature and reloading the search results based on the current filter, as shown below.

|  |
| --- |
| C# |
| * + 1. async void Filter\_Checked(object sender, RoutedEventArgs e)     2. {     3. var filter = (sender as FrameworkElement).DataContext;     4. // Mirror the change into the CollectionViewSource.     5. // This is most likely not needed.     6. if (filtersViewSource.View != null)     7. {     8. filtersViewSource.View.MoveCurrentTo(filter);     9. }     10. // Determine what filter was selected     11. var selectedFilter = filter as Filter;     12. if (selectedFilter != null)     13. {     14. // Mirror the results into the corresponding Filter object to allow the     15. // RadioButton representation used when not snapped to reflect the change     16. selectedFilter.Active = true;     17. // Load the search results and filter them based on user-selected filter value     18. \_searchResults = await JobDataSource.SearchJobsBySearchTextAsync(\_queryText);     19. \_searchResults = (selectedFilter.Name.Equals("All")) ? \_searchResults :     20. new List<Job>(\_searchResults.Where(job => job.Status.Equals(selectedFilter.Name)).ToList());     21. this.DefaultViewModel["Results"] = \_searchResults;     22. // Ensure results are found     23. object results;     24. ICollection resultsCollection;     25. if (this.DefaultViewModel.TryGetValue("Results", out results) &&     26. (resultsCollection = results as ICollection) != null &&     27. resultsCollection.Count != 0)     28. {     29. VisualStateManager.GoToState(this, "ResultsFound", true);     30. return;     31. }     32. }     33. // Display informational text when there are no search results.     34. this.DefaultViewModel["ShowFilters"] = false;     35. VisualStateManager.GoToState(this, "NoResultsFound", true);     36. } |

Code Sample . Filter check event handler

1. Build and re-run the application. Perform any search and then click on any of the filter. You would notice that only specific number of filtered results are shown for each filter.
2. When a job item is clicked, the details page for that job should open. To enable this, add the following click event in the **GridView** definition of the SearchResultsPage.xaml file.

|  |
| --- |
| XAML |
| * + 1. ItemClick="ItemGridView\_OnItemClick" |

Code Sample . Item click event

1. The event handler has to be defined in SearchResultsPage.xaml.cs to navigate to job details page. Add the following code to the page.

|  |
| --- |
| C# |
| * + 1. /// <summary>     2. /// This method handles the OnItemClick event of the ItemGridView control.     3. /// On clicking on a product, the user is navigated to the details page for that product.     4. /// The Product ID is passed as a parameter.     5. /// </summary>     6. private void ItemGridView\_OnItemClick(object sender, ItemClickEventArgs e)     7. {     8. this.Frame.Navigate(typeof(JobDetailPage), (e.ClickedItem as Job).JobId);     9. } |

Code Sample . Item click event handler

### Task 5: Update Search Results Page Styles

Since this is a new page that we’ve added to the project, the proper styles haven’t yet been applied to it. Our final task in this exercise is to implement a style for the search results page that is consistent with the rest of the application.

1. Make the following changes to the parent grid. The idea is to use a consistent style that has already been defined in CustomStyles.xaml. Also, the height of the header row has to be reduced to 120, as shown below.

|  |
| --- |
| XAML |
| * + 1. <Grid Style="{StaticResource LayoutRootStyle}">     2. <Grid.RowDefinitions>     3. <RowDefinition Height="120"/>     4. <RowDefinition Height="\*"/>     5. </Grid.RowDefinitions>     6. …     7. </Grid> |

Code Sample . Define the styles for the parent grid

1. For the page title and back button styling, replace the existing code with the code below.

|  |
| --- |
| XAML |
| * + 1. <!-- Back button and page title -->     2. <Grid Background="{StaticResource PageHeaderBackgroundBrush}">     3. <Grid.ColumnDefinitions>     4. <ColumnDefinition Width="120"/>     5. <ColumnDefinition Width="Auto" />     6. <ColumnDefinition Width="Auto" />     7. <ColumnDefinition Width="\*"/>     8. </Grid.ColumnDefinitions>     9. <AppBarButton x:Name="backButton" Icon="Back" Height="95" Margin="10,28,10,0"     10. Command="{Binding NavigationHelper.GoBackCommand, ElementName=pageRoot}"     11. Visibility="{Binding IsEnabled, Converter={StaticResource BooleanToVisibilityConverter}, RelativeSource={RelativeSource Mode=Self}}"     12. AutomationProperties.Name="Back" AutomationProperties.AutomationId="BackButton"     13. AutomationProperties.ItemType="Navigation Button"/>     14. <TextBlock x:Name="pageTitle" Text="Search Results Page" Style="{StaticResource HeaderTextBlockStyle}" Grid.Column="1"     15. IsHitTestVisible="false" TextWrapping="NoWrap" VerticalAlignment="Bottom" Margin="0,0,30,40"/>     16. <TextBlock x:Name="resultText" Grid.Column="2" Text="Results for&#x00a0; " IsHitTestVisible="false" Style="{StaticResource SubheaderTextBlockStyle}" TextWrapping="NoWrap" VerticalAlignment="Bottom" Margin="0,0,0,40" />     17. <TextBlock x:Name="queryText" Grid.Column="3" Text="{Binding QueryText}" IsHitTestVisible="false" Style="{StaticResource SubheaderTextBlockStyle}" TextWrapping="NoWrap" VerticalAlignment="Bottom" Margin="0,0,0,40" />     18. </Grid> |

Code Sample . Applying page header styles

1. Now, a minor adjustment should be made for the filter list control to make it more appealing. Replace the existing code for the control ItemsControl with the code below.

|  |
| --- |
| XAML |
| * + 1. <ItemsControl     2. x:Name="filtersItemsControl"     3. Canvas.ZIndex="1"     4. Margin="116,20,120,30"     5. ItemsSource="{Binding Source={StaticResource filtersViewSource}}"     6. Visibility="{Binding ShowFilters, Converter={StaticResource BooleanToVisibilityConverter}}">     7. <ItemsControl.ItemsPanel>     8. <ItemsPanelTemplate>     9. <StackPanel Orientation="Horizontal"/>     10. </ItemsPanelTemplate>     11. </ItemsControl.ItemsPanel>     12. <ItemsControl.ItemTemplate>     13. <DataTemplate>     14. <RadioButton     15. Margin="0,0,30,0"     16. GroupName="Filters"     17. IsChecked="{Binding Active, Mode=TwoWay}"     18. Checked="Filter\_Checked"     19. Style="{StaticResource TextBlockButtonStyle}">     20. <TextBlock Text="{Binding Description}" TextWrapping="NoWrap" Margin="3,-7,3,10" Style="{StaticResource SubheaderTextBlockStyle}" />     21. </RadioButton>     22. </DataTemplate>     23. </ItemsControl.ItemTemplate>     24. </ItemsControl> |

Code Sample . Applying styles to the items control

1. Build and run the application and conduct a search using the Search charm. The final screen should appear as shown in figure below.



Figure . The completed search results page

## Exercise 4: Share

Like search, Windows Store apps expose share in a consistent fashion. Swiping on the right edge of the screen or pressing the Windows+C key combination invokes the charm bar. You can implement share as a default function for the app as a whole and/or customize it for specific content within the app. In the default mode of sharing, a screenshot of the current page is taken, which you can then email. Sharing specific content on a page requires customization of the code for the Share contract.

In this exercise you will learn how to implement the Share contract for the job details page of the Field Engineer application. You will enable the job details page to share information about the job (job number, ID, title, customer address, etc.) in a structured manner. The rest of the pages in the application will use default sharing behavior.

There are two tasks in this exercise:

1. Add handler for Share charm invocation.
2. Define the content structure to share.

### Task 1: Add Handler for Share Charm Invocation

Before we start this task, build and run the application, and then invoke the Share charm for any page. You will notice that the page shows the default behavior for sharing, capturing a screenshot of the page to be shared.

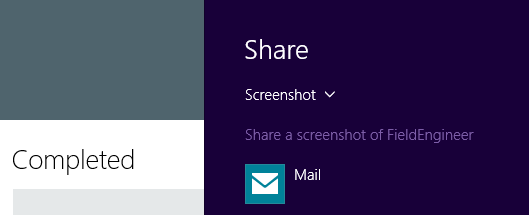


Figure . Default Share charm behavior

On selecting **Mail**, a sample email is created with the screenshot of the shared page as an attachment.

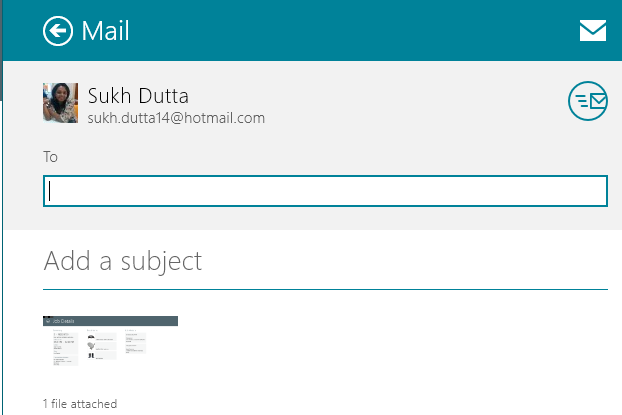


Figure . Sample mail with a screen capture of the current page as an attachment

Now, to customize share functionality for the job details page, the first task is to create an object of the **DataTransferManager,** which is the starting point of any sharing operation and defines the handler that handles the **DataRequested** event. The [**DataRequested**](http://msdn.microsoft.com/en-us/library/windows/apps/windows.applicationmodel.datatransfer.datatransfermanager.datarequested.aspx) event occurs when the user taps or clicks the Share charm.

1. First you need to add the right namespaces to your app so you can create and process the objects related to sharing. Add the following namespace to the JobDetails.xaml.cs page.

|  |
| --- |
| C# |
| * + 1. using Windows.ApplicationModel.DataTransfer; |

Code Sample . Add namespace to JobDetails.xaml.cs

1. The next step is to get an instance of the **DataTransferManager** and setup the handlers for the **DataRequested** event. In the JobDetailPage.xaml.cs page, update the **OnNavigatedTo** and the **OnNavigatedFrom** methods as shown below.

|  |
| --- |
| C# |
| * + 1. protected override void OnNavigatedTo(NavigationEventArgs e)  {      //Call the corresponding method on Navigation helper      navigationHelper.OnNavigatedTo(e);     2. //Get an instance of DataTransferManager and add handler for DataRequested event      DataTransferManager dataTransferManager = DataTransferManager.GetForCurrentView();      dataTransferManager.DataRequested += DataRequested;             } |

Code Sample . Add Handler for DataTransferManager.DataRequested event

|  |
| --- |
| C# |
| * + 1. protected override void OnNavigatedFrom(NavigationEventArgs e)  {      //Call the corresponding method on Navigation helper      navigationHelper.OnNavigatedFrom(e);     2. //Get an instance of DataTransferManager and remove handler for DataRequested event      DataTransferManager dataTransferManager = DataTransferManager.GetForCurrentView();      dataTransferManager.DataRequested -= DataRequested;             } |

Code Sample . Remove Handler for DataTransferManager.DataRequested event

1. Now, add the **DataRequested** handler to the page by adding the requisite code to the JobDetailPage.xaml.cs file, as shown below.

|  |
| --- |
| C# |
| * + 1. /// <summary>     2. /// This method contains the logic to prepare the data which would be shared. The job     3. /// details would be prepared in the form of a table which can then be shared through email.     4. /// </summary>     5. private void DataRequested(DataTransferManager sender, DataRequestedEventArgs args)     6. {     7. } |

Code Sample . Add DataRequested handler

1. Build and run the application, and then evoke the Share charm for job details page. You will notice that the page no longer shows the default sharing behavior. Instead it offers two options: the default involving a screenshot, and another labeled “Field Engineer,” as shown in figure below.

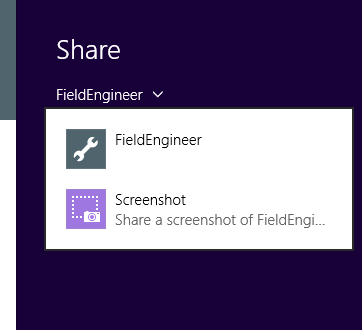


Figure . Sharing options for job details page

1. Select **Field Engineer**. As the handler we created has no code, you should see the message “There is nothing to share right now,” shown below.



Figure . Intermediate state of share without the content defined

### Task 2: Define the Content Structure to Share

Let’s solve this problem now by defining the actual content to be shared from the job details page.

1. Add the following code to the **DataRequested** handler. This transforms the content into HTML for sharing.

|  |
| --- |
| C# |
| * + 1. DataRequest request = args.Request;     2. request.Data.Properties.Title = "job details for " + job.JobNumber + " - " + job.JobId;     3. request.Data.Properties.Description = job.Title;     4. //Create table format using the job details     5. StringBuilder htmlcontent = new StringBuilder("<p> <Table style=\"font-family:Segoe UI;\"");     6. htmlcontent.Append("<tr><td> <b>Job Number: </b>" + job.JobNumber + " </td> </tr> ");     7. htmlcontent.Append("<tr><td> <b>Job ID: </b>" + job.JobId + " </td> </tr> ");     8. htmlcontent.Append("<tr><td> <b>Title: </b>" + job.Title + " </td> </tr> ");     9. htmlcontent.Append("<tr><td> <b>Scheduled For: </b>" + job.StartTime + " - " + job.EndTime +     10. " </td> </tr> ");     11. htmlcontent.Append("<tr> <td> <b> Job Status: </b>" + job.Status + "</td> </tr> <tr> <td> </td> </tr>");     12. htmlcontent.Append("<tr> <td> <b> Customer Address: </b> </td> </tr>");     13. htmlcontent.Append("<tr> <td> " + job.Customer.Prefix + " " + job.Customer.FirstName + " " +     14. job.Customer.Surname + " </td> </tr>");     15. htmlcontent.Append("<tr><td> " + job.Customer.HouseNumberOrName + ", " + job.Customer.Street +     16. " </td></tr>");     17. htmlcontent.Append("<tr><td> " + job.Customer.Town + " </td></tr>");     18. htmlcontent.Append("<tr><td> " + job.Customer.County + " </td></tr>");     19. htmlcontent.Append("<tr><td> " + job.Customer.Postcode + " </td></tr><tr><td></td></tr>");     20. htmlcontent.Append("<tr><td> <b> Customer Contact: </b></td></tr>");     21. htmlcontent.Append("<tr><td> " + job.Customer.PrimaryContactNumber + " </td></tr>");     22. htmlcontent.Append("<tr><td> " + job.Customer.AdditionalContactNumber + " </td></tr>");     23. htmlcontent.Append("</table> </p>");     24. request.Data.SetHtmlFormat(HtmlFormatHelper.CreateHtmlFormat(htmlcontent.ToString())); |

Code Sample . Create content in HTML format for sharing

NOTE ▶ In addition to HTML, you can share content through the Share contract using a variety of formats. Here, we have chosen HTML in order to show the data in a simple table format. For more information about choosing the appropriate data format per your requirements, refer to [the MSDN article](http://msdn.microsoft.com/en-us/library/windows/apps/hh871364.aspx).

1. Build and run the application and then evoke the Share charm for the job details page. You will notice that the two sharing options are now updated: you can either share a screenshot of the page or share the job details for the current job, as shown immediately below.



Figure . New options for sharing job details page after content definition

1. Click on the **Job Details**… option. The content to be shared in now neatly formatted and added in a sample e-mail, as shown in figure below.

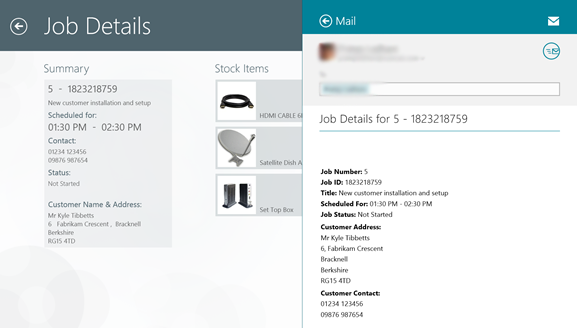


Figure . Sample mail with the content defined

## Exercise 5: App Bar

App bars are used to present secondary navigation, commands, and tools that can be hidden away when they aren't needed. You can put an app bar at the top of the page, at the bottom of the page, or both. App bars are hidden by default, and are shown or dismissed when the user right-clicks on a mouse, presses the Windows+Z key combination, or swipes from the top or bottom edge of the screen. They can also be shown programmatically when the user makes a selection or interacts with the app.

In this exercise we are going to add the app bar to the bottom of the equipment details page, and then add to it a navigational command button.

NOTE ▶ For more information on app bar controls in Windows 8.1, see the [MSDN Web site](http://msdn.microsoft.com/en-us/library/windows/apps/bg182878.aspx#AppBarControls). For detailed guidance on how to implement them optimally, refer to [the MSDN article](http://msdn.microsoft.com/en-us/library/windows/apps/hh465302.aspx).

### Task 1: Add Bottom App Bar

1. Open the EquipmentDetails.xaml file.
2. Append the following piece of XAML code inside the **Page** tag.

|  |
| --- |
| XAML |
| * + 1. <Page.BottomAppBar>     2. <CommandBar>     4. </CommandBar>     5. </Page.BottomAppBar> |

Code Sample . Adding an app bar in EquipmentDetails.xaml

### Task 2: Add Command to App Bar

The app bar in place, now the task is to add a command to it. For this exercise, we’ll add a navigation command button that takes the user back to the welcome page. In Windows 8.1, app bar commands are classified as either primary or secondary, depending on the importance of the command in the context of a given scenario. Since navigating to the jobs listing page is a secondary action for the page, we will add it to the **SecondaryCommands** collection.

1. Open the EquipmentDetails.xaml file.
2. Add the following code between the **CommandBar** tags.

|  |
| --- |
| XAML |
| * + 1. <Page.BottomAppBar>     2. <CommandBar>     3. <CommandBar.SecondaryCommands>     4. <AppBarButton x:Name="Home" Label="Home" Icon="Home" Click="Home\_Click"/>     5. </CommandBar.SecondaryCommands>     6. </CommandBar>     7. </Page.BottomAppBar> |

Code Sample . Add command button to the app bar

1. Open the EquipmentDetails.xaml.cs file and add the following code, which contains the event handler for the app bar button.

|  |
| --- |
| C# |
| * + 1. /// <summary>     2. /// Handles the App Bar event for the Home command.     3. /// </summary>     4. private void Home\_Click(object sender, RoutedEventArgs e)     5. {     6. // Navigate to job listing page     7. this.Frame.Navigate(typeof(JobListPage));     8. } |

Code Sample . Add event handler for app bar button

1. Run the application and navigate to the equipment details page.
2. Right-click the screen using your mouse (or swiping from top or bottom if you’re on a touchscreen). The app bar will be summoned from the bottom of the page, as shown below.

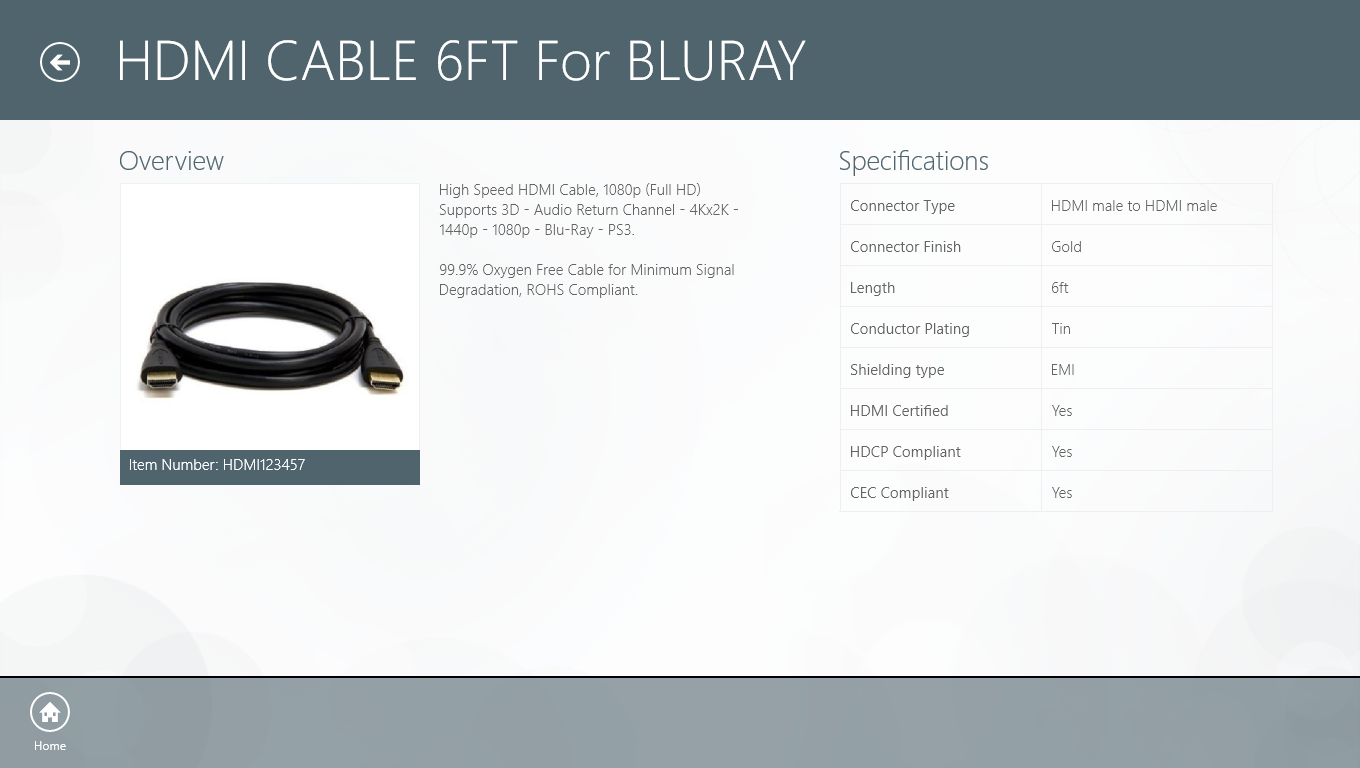


Figure . App bar with a navigational command button

## Exercise 6: Live Tiles

A tile is used to represent an app on the Start screen, and can be configured to present rich and engaging content to users when the app is not running. Tapping or clicking the tile launches the app.

Tiles can be live (updated through notifications) or you can leave them static. Tiles begin as a default tile, defined in the app's manifest. A static tile will always display the default content, which is generally a full-tile logo image. A live tile can update the default tile to show new content, but can return to the default if the update expires or is removed. A tile can also display a status badge, which can be a number—such as the quantity of emails or messages in queue—or a glyph.

In this exercise, you will learn how to implement a live tile for the Field Engineer application by completing three tasks:

1. Choose the type of live tile.
2. Configure the content for the live tile.
3. Enable the live tile.

### Task 1: Choose the Type of Live Tile

A Windows 8.1-based Windows Store app has four kinds of tiles (small, wide, medium, large), and of these, all except the small variety can be configured for live updates. Visual Studio 2013 also provides many formats for each kind of live tile, giving you many options for configuring your tile to look and behave the way you want.

For this lab, we will configure the live tile functionality for a wide tile. Among the wide tile formats, we want to show the content as two lines of text: with job number and ID on the first line, and the job title on the second line.

1. Open the App.xaml.cs file.
2. Add the method shown in the code sample below.

|  |
| --- |
| C# |
| * + 1. /// <summary>     2. /// This method takes a Job as its input and generates an XML Document which contains the /// actual values to be displayed in the live tile.     3. /// </summary>     4. private XmlDocument CreateWideTile(Job job)     5. {     6. // Create a live update for a wide tile     7. XmlDocument wideTileXml = TileUpdateManager.GetTemplateContent( TileTemplateType.TileWide310x150Text01);     8. // Assign text     9. XmlNodeList wideTileTextAttributes = wideTileXml.GetElementsByTagName("text");     10. wideTileTextAttributes[0].AppendChild(wideTileXml.CreateTextNode(job.JobNumber + " - " + job.JobId));     11. wideTileTextAttributes[1].AppendChild(wideTileXml.CreateTextNode(job.Title));     12. return wideTileXml;     13. } |

Code Sample . Generate XML document for value to be displayed in live tile

### Task 2: Configure the Content for the Live Tile

Now that we have decided how our live tile should look, we need to fetch the data to be displayed in the live tile and get the XML schema for the data.

1. Add the following method in JobDataSource.cs, which returns the list of pending jobs for an engineer to see in the app dashboard.

|  |
| --- |
| C# |
| * + 1. /// <summary>     2. /// This method gets all the jobs which are still pending (jobs with status 'Not Started').     3. /// </summary>     4. /// <returns>List of Pending Jobs</returns>     5. public static async Task<List<Job>> GetPendingJobsAsync()     6. {     7. await \_jobDataSource.ReadXmlDataFromLocalStorageAsync();     8. var matches = \_jobDataSource.AllJobs.Where((item) => item.Status == Constants.PendingStatus).ToList();     9. return matches;     10. } |

Code Sample . Add pending jobs method

1. In the App.xaml.cs file, add the following code to fetch the list of pending jobs, cycle through them, and create the XML document for each job.

|  |
| --- |
| C# |
| * + 1. /// <summary>     2. /// This method creates a new notification for each job by iterating through a list of     3. /// pending jobs and passes the notification to the TileNotificationManager.     4. /// It then enables notification for the application.     5. /// </summary>     6. private async void CreateLiveTiles()     7. {     8. var pendingJobs = await JobDataSource.GetPendingJobsAsync();     9. foreach (var job in pendingJobs)     10. {     11. // Create a tile notification.     12. TileNotification tileNotification = new TileNotification(CreateWideTile(job));     13. TileUpdateManager.CreateTileUpdaterForApplication().Update(tileNotification);     14. }     15. } |

Code Sample . Create an XML document for each job

### Task 3: Enable the Live Tile

1. Add the following code at the end of the **OnLaunched** method of the App.xaml.cs file.

|  |
| --- |
| C# |
| * + 1. // Call the method to create Live tiles     2. CreateLiveTiles(); |

Code Sample . Call the method to create live tiles

1. Add the following code at the end of the **CreateLiveTiles** method of the App.xaml.cs file. This line of code enables the tile to queue up to five notifications.

|  |
| --- |
| C# |
| * + 1. TileUpdateManager.CreateTileUpdaterForApplication().EnableNotificationQueue(true); |

Code Sample . Enable tile to queue notifications

1. Build and run the application and then navigate to your Windows Start screen. You should be able to see the Field Manager wide tile flipping through the list of pending jobs, as shown in the figure below.



Figure . Windows Start screen showing the live tile for the Field Engineer app

# Optional Exercises

## Exercise 1: Semantic Zoom

Semantic Zoom is a touch-optimized technique used by Windows Store apps for presenting and navigating large sets of related data or content within a single view. Semantic Zoom uses two distinct modes of classification (or zoom levels) for organizing and presenting the content: one low-level (or zoomed-in) mode that is typically used to display items in a flat, all-up structure and another, high-level (or zoomed-out) mode that displays items in groups and enables a user to quickly navigate and browse through the content.

This exercise will enable you to create a zoomed-out view on the list of jobs, which will, at a high level, show the 3 groups of jobs (Current, Completed, and Pending).

There are two tasks in this exercise:

1. Add Semantic Zoom control.
2. Define the **ZoomedOutView**.

### Task 1: Add Semantic Zoom Control

This task will enable you to define a Semantic Zoom control with your original **GridView** as the zoomed-in section.

1. Open the JobListPage.xaml file and add the following lines of code.

|  |
| --- |
| XAML |
| * + 1. <SemanticZoom Grid.RowSpan="2" x:Name="SemanticZoom">     2. <SemanticZoom.ZoomedInView>     3. </SemanticZoom.ZoomedInView>     4. </SemanticZoom> |

Code Sample . Add the Semantic Zoom control

1. Move **itemGridView** inside the **SemanticZoom.ZoomedInView** tag.

|  |
| --- |
| XAML |
| * + 1. <SemanticZoom Grid.RowSpan="2" x:Name="SemanticZoom">     2. <SemanticZoom.ZoomedInView>     3. <!-- Horizontal Scrolling GridView -->     4. <GridView     5. x:Name="itemGridView"     6. AutomationProperties.AutomationId="ItemGridView"     7. AutomationProperties.Name="Grouped Items"     8. …     9. </GridView>     10. </SemanticZoom.ZoomedInView>     11. </SemanticZoom> |

Code Sample . Move GridView inside ZoomedInView

### Task 2: Define ZoomedOutView

For **ZoomedOutView** we will be showing a high-level list of the group names and the number of items in each group.

1. Check to see if the style **JobGroupItemTemplate** is present in the CustomTemplates.xaml file. If it is not, add the following code to the file CustomTemplates.xaml.

|  |
| --- |
| XAML |
| * + 1. <DataTemplate x:Key="JobGroupItemTemplate">     2. <Grid Height="280" Width="240" >     3. <Grid.Background>     4. <SolidColorBrush Color="{StaticResource TileColor}" Opacity="0.3"/>     5. </Grid.Background>     6. <TextBlock Text="{Binding Group.Jobs.Count}" VerticalAlignment="Top" TextAlignment="Right" TextWrapping="NoWrap" Margin="0,0,20,0" FontSize="104" FontWeight="Light" Foreground="{StaticResource ApplicationForegroundThemeBrush}" />     7. <TextBlock Text="{Binding Group.Title}" TextAlignment="Left" VerticalAlignment="Bottom" Margin="15" Style="{StaticResource SubheaderTextBlockStyle}" />     8. </Grid>     9. </DataTemplate> |

Code Sample . Item template for zoomed-out GridView items

1. Add the following XAML code to JobListPage.xaml after the **SemanticZoom.ZoomedOutView** closing tag. This is for creating the zoomed-out view.

|  |
| --- |
| XAML |
| * + 1. <SemanticZoom.ZoomedOutView>     2. <GridView x:Name="ZoomedOutGrid"     3. ItemTemplate="{StaticResource JobGroupItemTemplate}" >     4. <GridView.ItemsPanel>     5. <ItemsPanelTemplate>     6. <StackPanel VerticalAlignment="Center" Orientation="Horizontal" Margin="116,136,40,46"/>     7. </ItemsPanelTemplate>     8. </GridView.ItemsPanel>     9. </GridView>     10. </SemanticZoom.ZoomedOutView> |

Code Sample . Zoomed-out view

1. The last step is to assign the data to the **ItemsSource** of the zoomed-out **GridView**. This uses the same data as the zoomed-in view but in a more grouped manner. Copy the following code into the **LoadState** method of the JoblistPage.xaml.cs file.

|  |
| --- |
| C# |
| * + 1. // Assign the groups as the data source for Zoomed Out view     2. (SemanticZoom.ZoomedOutView as ListViewBase).ItemsSource = groupedItemsViewSource.View.CollectionGroups; |

Code Sample . Assign data source to the zoomed-out view

1. Build and run the application. You should see a minus symbol (“-“) at the bottom right corner of the screen. Click it if you are using a mouse, or use a pinch gesture in a touchscreen environment. It should now be zoomed-out, as shown below.

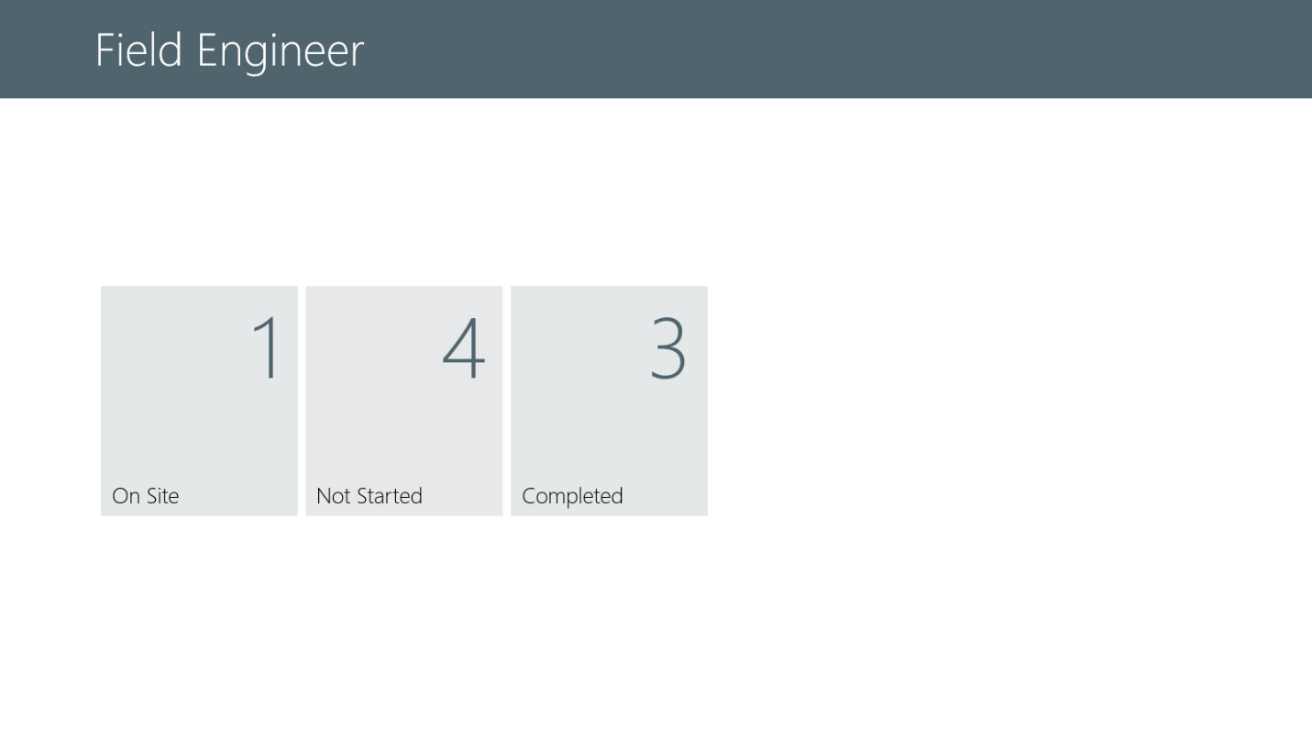


Figure . Zoomed-out view in the Field Engineer app

1. Click any of the tiles. The screen should zoom in to that specific set of jobs.

## Exercise 2: Windowing Modes

In Windows 8.1, there is no fixed-width view state. Instead, users can now resize apps continuously down to a minimum width. (The default minimum width of an app is 500 pixels.) So apps no longer have the snapped and filled view states. Instead, you develop your app to be functional and good-looking at any size down to the minimum.

This exercise will show you how to create narrower views for the jobs list page, which you can resize down to a defined minimum width.

This exercise includes four tasks

1. Define the minimum width for the application.
2. Define the narrower view for the page.
3. Set up the visual states.
4. Handle the **Window.SizeChanged** event.

### Task 1: Define the Minimum Width for the Application

In this task, you will define the minimum width to which the window can be resized.

1. Open the Package.appxmanifest file from the Solution Explorer and then select the **Application UI** tab.
2. Find the **Minimum width** box, which is not set by default. Change the option to 320 px. Now your application can be narrowed down to 320 px.

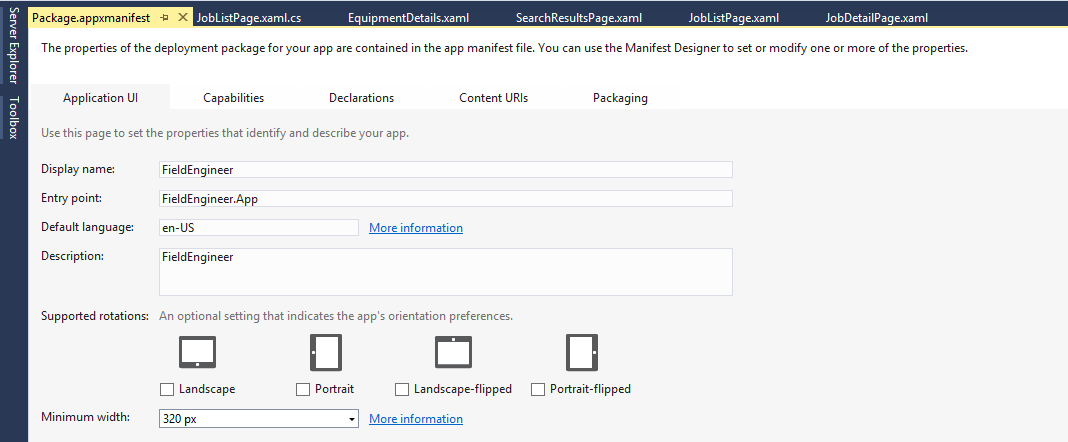


Figure . Changing the minimum window width in the manifest

### Task 2: Define the Narrower View for the Page

1. Consider the structure and layout of the content beyond a specific width. It may make more sense to display job cards as a vertical scrolling list when the width of the page becomes narrower than 620 px.
2. Add the following XAML code to JobListPage.xaml inside the main layout grid. This is to define the structure of a **ListView** control containing the job cards when the page is snapped.

|  |
| --- |
| XAML |
| * + 1. <!--     2. ======================================================================     3. DEFINE SECONDARY VIEW:     5. This is the narrow view for the page. This view is used when the page     6. is resized beyond a specific width. This view represents the vertical     7. listing of the job items using ListView.     8. ======================================================================     9. -->     10. <ListView     11. x:Name="itemListView"     12. AutomationProperties.AutomationId="ItemListView"     13. AutomationProperties.Name="Grouped Items"     14. Grid.Row="1"     15. Visibility="Collapsed"     16. ItemsSource="{Binding Source={StaticResource groupedItemsViewSource}}"     17. ItemTemplate="{StaticResource JobItemTemplate}"     18. SelectionMode="None"     19. IsSwipeEnabled="false"     20. IsItemClickEnabled="True">     21. <ListView.ItemsPanel>     22. <ItemsPanelTemplate>     23. <StackPanel Orientation="Vertical" Margin="16,20,16,40"/>     24. </ItemsPanelTemplate>     25. </ListView.ItemsPanel>     26. <ListView.ItemContainerStyle>     27. <Style TargetType="ListViewItem">     28. <Setter Property="Margin" Value="0" />     29. </Style>     30. </ListView.ItemContainerStyle>     31. <ListView.GroupStyle>     32. <GroupStyle>     33. <GroupStyle.HeaderTemplate>     34. <DataTemplate>     35. <TextBlock Text="{Binding Title}" Margin="4,0,0,10" Style="{StaticResource SubheaderTextBlockStyle}" />     36. </DataTemplate>     37. </GroupStyle.HeaderTemplate>     38. <GroupStyle.Panel>     39. <ItemsPanelTemplate>     40. <VariableSizedWrapGrid Orientation="Horizontal" Margin="0,0,0,10"/>     41. </ItemsPanelTemplate>     42. </GroupStyle.Panel>     43. </GroupStyle>     44. </ListView.GroupStyle>     45. </ListView> |

Code Sample . ListView control for a narrow, “snapped” view

### Task 3: Set Up the Visual States

Now, the task is to handle the various visual states (normal, narrow etc.) and define how the visual elements on your page would appear during each state.

1. Check to see how the page responds on resizing. The horizontal scrolling **GridView** should be invisible when the application is resized below 620 px. At this width, the **ListView** that you added in the previous task should become visible.
2. Add the following XAML code to your page, JobListPage.xaml, inside the main layout grid. This is for handling the visual states.

|  |
| --- |
| XAML |
| * + 1. <!-- ===================================== -->     2. <!-- VISUAL STYLES -->     3. <!-- This is the narrow view for the page. -->     4. <!-- ===================================== -->     5. <VisualStateManager.VisualStateGroups>     6. <VisualStateGroup x:Name="LayoutStates">     7. <VisualState x:Name="PrimaryLayout">     8. <Storyboard></Storyboard>     9. </VisualState>     10. <VisualState x:Name="NarrowLayout">     11. <Storyboard>     12. <Storyboard>     13. <ObjectAnimationUsingKeyFrames Storyboard.TargetName="pageTitle" Storyboard.TargetProperty="Margin">     14. <DiscreteObjectKeyFrame KeyTime="0" Value="-100,0,20,40"/>     15. </ObjectAnimationUsingKeyFrames>     16. <ObjectAnimationUsingKeyFrames Storyboard.TargetName="pageTitle" Storyboard.TargetProperty="FontSize">     17. <DiscreteObjectKeyFrame KeyTime="0" Value="48"/>     18. </ObjectAnimationUsingKeyFrames>     19. <ObjectAnimationUsingKeyFrames Storyboard.TargetName="itemListView" Storyboard.TargetProperty="Visibility">     20. <DiscreteObjectKeyFrame KeyTime="0" Value="Visible"/>     21. </ObjectAnimationUsingKeyFrames>     22. <ObjectAnimationUsingKeyFrames Storyboard.TargetName="SemanticZoom" Storyboard.TargetProperty="Visibility">     23. <DiscreteObjectKeyFrame KeyTime="0" Value="Collapsed"/>     24. </ObjectAnimationUsingKeyFrames>     25. </Storyboard>     26. </Storyboard>     27. </VisualState>     28. </VisualStateGroup>     29. </VisualStateManager.VisualStateGroups> |

Code Sample . Setting up visual states for the page

### Task 4: Handle the Window.SizeChanged Event

The last task in this exercise involves handling the event that occurs whenever you change the width of the page. This event must be handled and the visual state must be selected accordingly.

1. Add the following code to the constructor in the page JobListPage.xaml.cs.

|  |
| --- |
| C# |
| * + 1. // Register a handler for the Window.SizeChanged event     2. Window.Current.SizeChanged += Window\_SizeChanged; |

Code Sample . Register a handler for the Window.SizeChanged event

1. Add the following method for changing visual states when the width of the page is made less than 620 px.

|  |
| --- |
| C# |
| * + 1. /// <summary>     2. /// Check the window width and update the Visual State     3. /// </summary>     4. void UpdateVisualState (double width)     5. {     6. if (width < 620)     7. {     8. VisualStateManager.GoToState(this, "NarrowLayout", true);     9. }     10. else     11. {     12. VisualStateManager.GoToState(this, "PrimaryLayout", true);     13. }     14. }     15. /// <summary>     16. /// Represents the event handler for the Window.SizeChanged event.     17. /// This event is raised whenever the window is resized and helps     18. /// handle any changes to visual state of the page.     19. /// </summary>     20. void Window\_SizeChanged(object sender, WindowSizeChangedEventArgs e)     21. {   // Check the window size and update the Visual State   * + 1. UpdateVisualState(e.Size.Width);     2. } |

Code Sample . Method for changing visual states

1. Update the LoadState method to include the method call to **UpdateVisualState** method, as shown the in the code snippet below. This will ensure that the visual state is updated even during page load.

|  |
| --- |
| C# |
| * + 1. private async void navigationHelper\_LoadState(object sender, LoadStateEventArgs e)     2. {     3. ...     4. // Check the window size and update the Visual State     5. UpdateVisualState(Window.Current.Bounds.Width);     6. } |

Code sample . LoadState method with call to UpdateVisualState

1. Build and run the application. The application will be in the default view. Drag and narrow the application to half the screen. It will still have the horizontal view, as shown below in Figure 29.

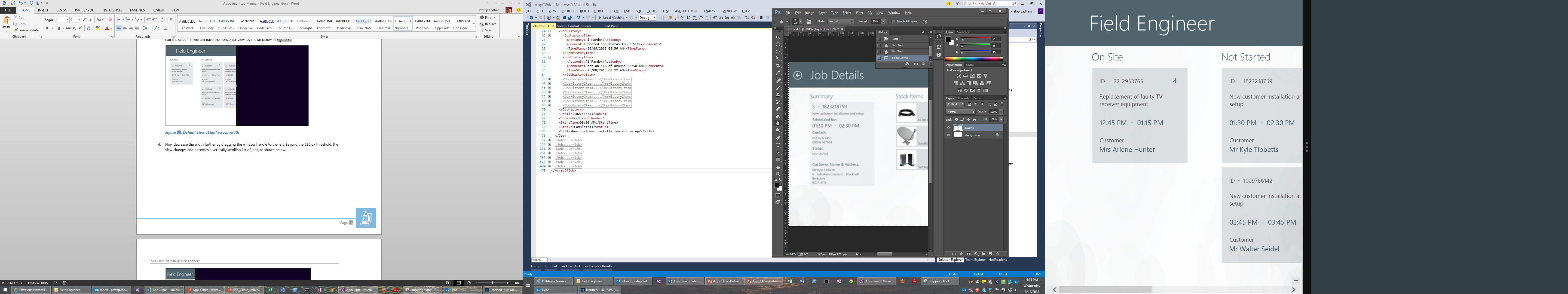


Figure . Default view at half screen width

1. Now decrease the width further by dragging the window handle to the left. Beyond the 620 px threshold, the view changes and becomes a vertically scrolling list of jobs, as shown below.

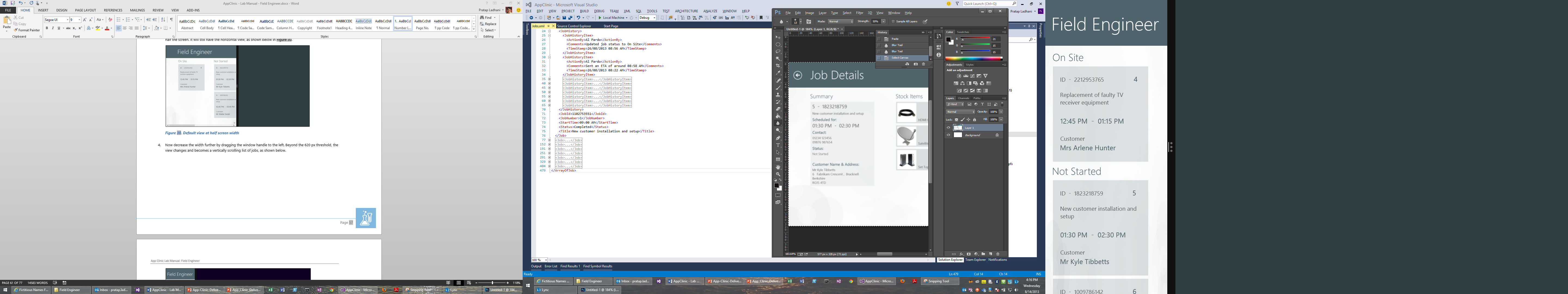


Figure . Vertical scrolling list in narrower view

## Exercise 3: Secondary Tiles

In addition to the primary tile (see above, *Exercise 6: Live Tiles*), a Windows Store app can also be configured to enable one or more secondary tiles. This features permits the pinning of specific, non-default sections of the application on the Windows Start screen so they can be accessed directly without having to go through the steps of navigating via the default entry point of the app. For example, a user may wish to pin to the Start screen a product page, article, or specific content hub of a given app for convenient reference later.

In this exercise, you will enable secondary tiles for your application so that its users may select any particular job and pin it to their Start screens. By clicking (or tapping, as the case may be) the secondary tile, your users will directly access the job details page for the pinned job.

In this exercise, there are five tasks:

1. Add the **Pin To Start** button.
2. Handle the button event.
3. Enable secondary tile pinning.
4. Enable secondary tile unpinning.
5. Enable entry through the secondary tile.

### Task 1: Add the Pin To Start Button

1. Open the JobDetailPage.xaml file.
2. Append the following piece of XAML code inside Page tag.

|  |
| --- |
| XAML |
| * + 1. <Page.BottomAppBar>     2. <CommandBar>     3. <CommandBar.SecondaryCommands>     4. <AppBarButton x:Name="PinToStart" Click="PinToStart\_Click" Loaded="PinToStart\_Loaded"/>     5. </CommandBar.SecondaryCommands>     6. </CommandBar>     7. </Page.BottomAppBar> |

Code Sample . Adding the app bar with PinToStart button in JobDetailPage.xaml

### Task 2: Handle the Button Event

The **Pin To Start** button has a click event that needs a handler.

1. Add the following code in the JobDetails.xaml.cs file.

|  |
| --- |
| C# |
| * + 1. /// <summary>     2. /// Handles the App Bar event - Pin To Start. This event is used to pin the current Job as a /// secondary tile on the Start screen     3. /// </summary>     4. private async void PinToStart\_Click(object sender, RoutedEventArgs e)     5. {     6. await SecondaryTileCreation(sender);     7. } |

Code Sample . Adding the handler for the click event

1. The **SecondaryTileCreation** method checks if a secondary tile for the specific job is already present. If so, then the user is given the option to unpin the tile from the Start screen. Otherwise, the user can pin the job. Add the following code in the JobDetails.xaml.cs file.

|  |
| --- |
| C# |
| * + 1. /// <summary>     2. /// This method checks if a secondary tile is already present for this job. If it is already /// present, the job is unpinned, if not the job is pinned.     3. /// </summary>     4. private async Task SecondaryTileCreation(object sender)     5. {     6. string appbarTileId = job.JobId;     7. if (!SecondaryTile.Exists(appbarTileId))     8. {     9. await PinSecondaryTile(sender, appbarTileId);     10. }     11. else     12. {     13. //await UnpinSecondaryTile(sender, appbarTileId);     14. }     15. } |

Code Sample . Add method to check pin state for a given job

### Task 3: Enable Secondary Tile Pinning

To create a secondary tile, a sample tile element (or placeholder) must be created to show the user what the secondary tile will look like when pinned.

1. Add the following code in the JobDetails.xaml.cs file.

|  |
| --- |
| C# |
| * + 1. /// <summary>     2. /// This method creates the placeholder for the secondary tile and shows it above the 'Pin' button.     3. /// </summary>     4. /// <param name="element">The element.</param>     5. /// <returns></returns>     6. public static Rect GetElementRect(FrameworkElement element)     7. {     8. GeneralTransform buttonTransform = element.TransformToVisual(null);     9. Point point = buttonTransform.TransformPoint(new Point());     10. return new Rect(point, new Size(element.ActualWidth, element.ActualHeight));     11. } |

Code Sample . Create secondary tile placeholder

1. To fully enable pinning, you also need to make sure that the **Pin To Start** button functions as expected: once a job is pinned, the button should toggle from pin to unpin. Copy the following lines of code into the JobDetails.xaml.cs file to enable this functionality.

|  |
| --- |
| C# |
| * + 1. /// <summary>     2. /// This method assigns the style to the app bar button.     3. /// </summary>     4. /// <param name="showPinButton">if set to <c>true</c> [show pin button].</param>     5. private void ToggleAppBarButton(bool showPinButton, AppBarButton pinToStart)     6. {     7. if (pinToStart != null)     8. {     9. pinToStart.Icon = (showPinButton) ? new SymbolIcon(Symbol.Pin) :     10. new SymbolIcon(Symbol.UnPin);     11. pinToStart.Label = (showPinButton) ? "Pin To Start" : "Unpin";     12. }     13. } |

Code Sample . Enable toggling of the Pin To Start button

1. Next, we need to specify the method that actually pins a job to the Start screen. Copy the following code into the JobDetails.xaml.cs file.

|  |
| --- |
| C# |
| * + 1. /// <summary>     2. /// This method pins the secondary tile. The secondary tile is created using the required /// parameters and pinned. The user is shown a message informing whether the tile is pinned /// successfully.     3. /// </summary>     4. private async Task PinSecondaryTile(object sender, string appbarTileId)     5. {     6. // Prepare package images for use as the Tile Logo in our tile to be pinned     7. Uri smallLogo = new Uri("ms-appx:///Assets/Tile\_150X150.png");     8. Uri wideLogo = new Uri("ms-appx:///Assets/Tile\_310X150.png");     9. // Create a Secondary tile     10. string tileActivationArguments = appbarTileId;     11. string subTitle = job.JobNumber + " - " + job.JobId;     12. SecondaryTile secondaryTile = new SecondaryTile(appbarTileId, "Field Engineer - Job " +     13. job.JobNumber, subTitle,     14. tileActivationArguments,     15. TileOptions.ShowNameOnLogo | TileOptions.ShowNameOnWideLogo,     16. smallLogo, wideLogo);     17. secondaryTile.ForegroundText = ForegroundText.Light;     18. bool isPinned = await secondaryTile.RequestCreateForSelectionAsync( GetElementRect((FrameworkElement)sender),Windows.UI.Popups.Placement.Above);     19. if (isPinned)     20. {     21. MessageDialog dialog = new MessageDialog("Job " + job.JobNumber + " (" + job.Title + ") successfully pinned.");     22. await dialog.ShowAsync();     23. }     24. else     25. {     26. MessageDialog dialog = new MessageDialog("Job " + job.JobNumber + " (" + job.Title + ") not pinned.");     27. await dialog.ShowAsync();     28. }     29. ToggleAppBarButton(!isPinned, sender as AppBarButton);     30. } |

Code Sample . Method for pinning a secondary tile to the Start screen

1. Now we need to add the **Loaded** event for the app bar button that executes when the button has completed loading. Put this event handler in JobDetailPage.xaml.cs

|  |
| --- |
| C# |
| * + 1. /// <summary>     2. /// This method is called when the PinToStart button has completed loading     3. /// </summary>     4. /// <param name="sender"></param>     5. /// <param name="e"></param>     6. private void PinToStart\_Loaded(object sender, RoutedEventArgs e)     7. {     8. ToggleAppBarButton(!SecondaryTile.Exists(job.JobId), sender as AppBarButton);     9. } |

Code Sample . Event handler for Loaded event of Pin To Start button

1. Now it’s time to test it out. Build and run the application and bring up the app bar on the job details page by right-clicking on the page (or swiping from the upper or lower edge, as the case may be).

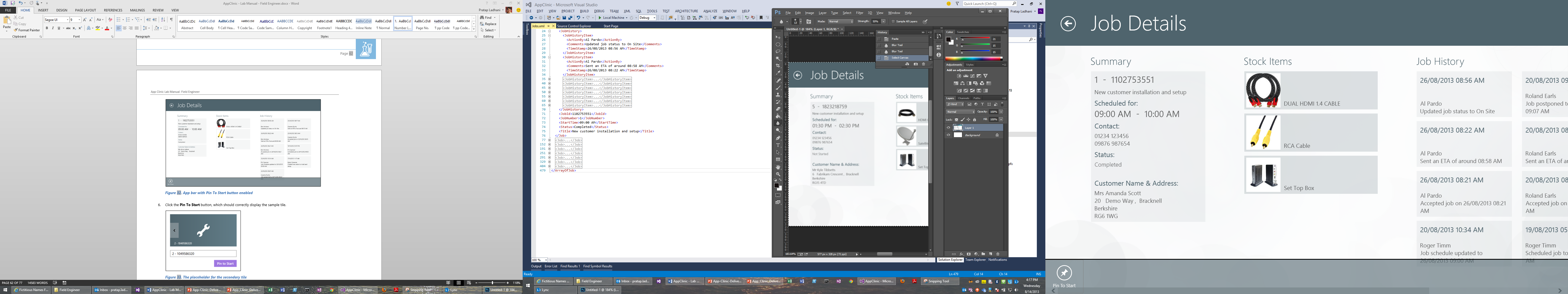


Figure . App bar with Pin To Start button enabled

1. Click the **Pin To Start** button, which should correctly display the sample tile.

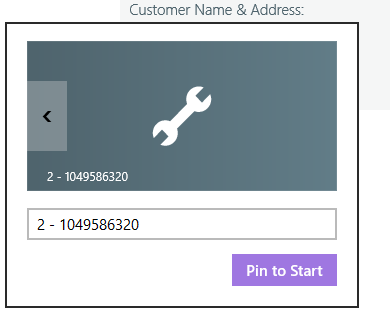


Figure . The placeholder for the secondary tile

### Task 4: Enable Secondary Tile Unpinning

Now you need to make it possible to unpin a tile that had been previously pinned to the Start screen.

1. Uncomment the call to the unpin method in the **SecondaryTileCreation** method in the JobDetails.xaml.cs file, as shown below.

|  |
| --- |
| C# |
| * + 1. /// <summary>     2. /// This method checks if a secondary tile is already present for this job. If it is already     3. /// present, the job is unpinned, if not the job is pinned.     4. /// </summary>     5. private async Task SecondaryTileCreation(object sender)     6. {     7. string appbarTileId = job.JobId;     8. if (!SecondaryTile.Exists(appbarTileId))     9. {     10. await PinSecondaryTile(sender, appbarTileId);     11. }     12. else     13. {     14. await UnpinSecondaryTile(sender, appbarTileId);     15. }     16. } |

Code Sample . Call the unpin method

1. Then add the code needed to unpin the secondary tile from the Start screen.

|  |
| --- |
| C# |
| * + 1. /// <summary>     2. /// This method unpins the existing secondary tile.     3. /// The user is showna message informing whether the tile is unpinned successfully     4. /// </summary>     5. private async Task UnpinSecondaryTile(object sender, string appbarTileId)     6. {     7. SecondaryTile secondaryTile = new SecondaryTile(appbarTileId);     8. bool isUnpinned = await secondaryTile.RequestDeleteForSelectionAsync(GetElementRect((FrameworkElement)sender), Windows.UI.Popups.Placement.Above);     9. if (isUnpinned)     10. {     11. MessageDialog dialog = new MessageDialog("Job " + job.JobNumber + " (" + job.Title + ") successfully unpinned.");     12. await dialog.ShowAsync();     13. }     14. else     15. {     16. MessageDialog dialog = new MessageDialog("Job " + job.JobNumber + " (" + job.Title + ") not unpinned.");     17. await dialog.ShowAsync();     18. }     19. ToggleAppBarButton(isUnpinned, sender as AppBarButton);     20. } |

Code Sample . Unpin a job from the Start screen

1. Run the application and select a job that’s already been pinned. Right-click to evoke the app bar. You should see that the style of the button has changed along with the text label.

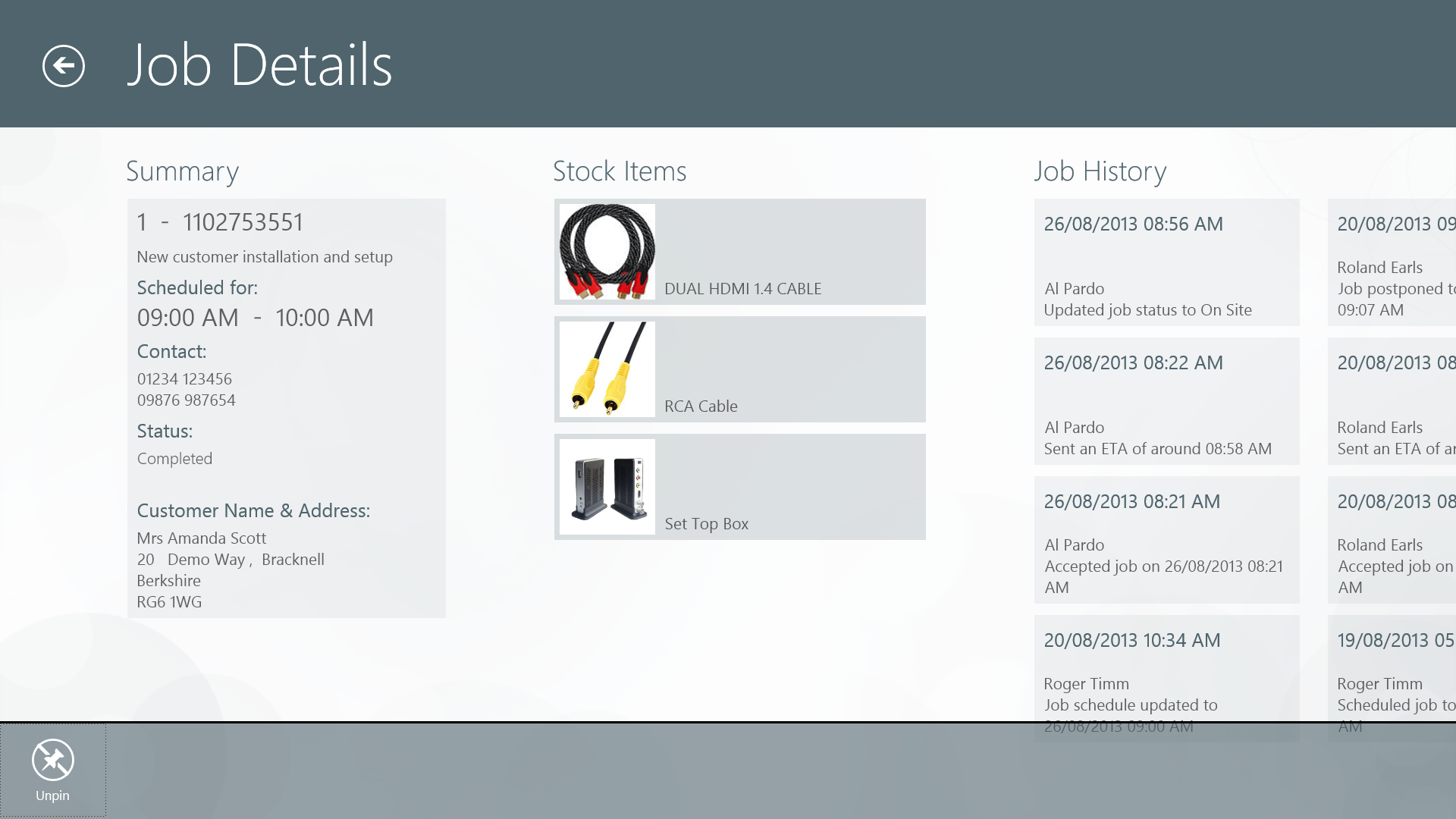


Figure . App bar now showing an Unpin button

1. Click the **Unpin** button. It will now display the tile you have selected for unpinning. Confirm by clicking the **Unpin from Start** button.

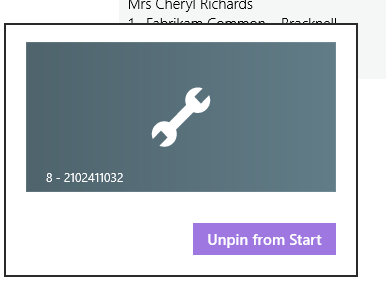


Figure . Image of the tile selected for unpinning

### Task 5: Enable Entry Through the Secondary Tile

1. To enable entering the application through the secondary tile, you need to replace code in the **OnLaunched** method of the App.xaml.cs file (Code Sample 61) with custom code (Code Sample 62).

|  |
| --- |
| C# |
| if (rootFrame.Content == null)  {      // When the navigation stack isn't restored navigate to the first page,      // configuring the new page by passing required information as a navigation      // parameter      if (!rootFrame.Navigate(typeof(JobListPage), e.Arguments))      {          throw new Exception("Failed to create initial page");      }  } |

Code Sample . Code to replace in the OnLaunched method of App.xaml.cs

|  |
| --- |
| C# |
| * + 1. if (rootFrame.Content == null || !string.IsNullOrEmpty(e.Arguments))  {      // When the navigation stack isn't restored navigate to the first page,      // configuring the new page by passing required information as a navigation      // parameter      if (string.IsNullOrEmpty(e.Arguments))      {          if (!rootFrame.Navigate(typeof(JobListPage), e.Arguments))          {              throw new Exception("Failed to create initial page");          }      }      else      {          if (!rootFrame.Navigate(typeof(JobDetailPage), e.Arguments))          {              throw new Exception("Failed to create initial page");          }      }  } |

Code Sample . Replacement code enabling secondary tile entry

1. Finally, check your work. Go to the Start screen and click on a secondary tile you created. It will open the job details page for the pinned job.



Figure . Secondary tile on the Start screen

# Conclusion

## Lab Completion

### Achievement Summary

Congratulations! You’ve just learned the fundamentals of building a Windows Store app, from how you bind data to the app and customize its look-and-feel with your branding, to the steps involved in implementing several key features, including:

* Search
* Share
* App bar
* Live tiles

#### The flexibility to develop what you need, the way you want

These lab exercises have been kept relatively short and simple on purpose so that you can more easily understand the basic structures and techniques of Windows Store app development before you turn to the task of building the next “killer app” for your business.

But you may also agree after working with it that, even at its most basic, the Windows Store-style app affords a great deal of variety in expression, suited to a wide range of scenarios, while maintaining design and functional consistency to deliver a better overall user experience. And although we chose C# and XAML for this lab, how you prefer to develop within the Windows 8 framework is up to you—whether using JavaScript with HTML/CSS; C#, VB, or C++ with XAML; or C++ with DirectX. The choice is yours.

So if you need to improve field operations, find a better way of presenting your catalogue of products to your customers, help your sales associates be more efficient in managing their clients, or wish to solve some other business problem entirely through the power of software, the skills you’ve acquired today plus the ones you brought with you can provide the foundation for imagining and creating innovative solutions based on Windows 8/8.1 that will serve to differentiate your business—and enhance your career.

Thank you for participating today and good luck!

# Appendix

## Table of Figures

[Figure 1. The structure of the Field Engineer solution 6](#_Toc367110241)

[Figure 2. Simple wireframe showing how the hierarchical navigation system works 7](#_Toc367110242)

[Figure 3. Welcome page for the Field Engineer app 8](#_Toc367110243)

[Figure 4. Class diagram of the Field Engineer data model 13](#_Toc367110244)

[Figure 5. Folders and files related to data binding 17](#_Toc367110245)

[Figure 6. Contents of the Assets folder 20](#_Toc367110246)

[Figure 7. Configure image asset sizes 21](#_Toc367110247)

[Figure 8. Data template for job items 24](#_Toc367110248)

[Figure 9. Data template for stock items 25](#_Toc367110249)

[Figure 10. Data template for job history items 26](#_Toc367110250)

[Figure 11. Search charm Scope Selector 27](#_Toc367110251)

[Figure 12. Search declaration 28](#_Toc367110252)

[Figure 13. Search charm scope selector with app now showing 28](#_Toc367110253)

[Figure 14. Add search results page 29](#_Toc367110254)

[Figure 15. Search results page (default view) 30](#_Toc367110255)

[Figure 16. Search results page 33](#_Toc367110256)

[Figure 17. Search results page with the proper template 34](#_Toc367110257)

[Figure 18. The completed search results page 37](#_Toc367110258)

[Figure 19. Default Share charm behavior 38](#_Toc367110259)

[Figure 20. Sample mail with a screen capture of the current page as an attachment 38](#_Toc367110260)

[Figure 21. Sharing options for job details page 40](#_Toc367110261)

[Figure 22. Intermediate state of share without the content defined 40](#_Toc367110262)

[Figure 23. New options for sharing job details page after content definition 41](#_Toc367110263)

[Figure 24. Sample mail with the content defined 42](#_Toc367110264)

[Figure 25. App bar with a navigational command button 44](#_Toc367110265)

[Figure 26. Windows Start screen showing the live tile for the Field Engineer app 47](#_Toc367110266)

[Figure 27. Zoomed-out view in the Field Engineer app 51](#_Toc367110267)

[Figure 28. Changing the minimum window width in the manifest 52](#_Toc367110268)

[Figure 29. Default view at half screen width 56](#_Toc367110269)

[Figure 30. Vertical scrolling list in narrower view 56](#_Toc367110270)

[Figure 31. App bar with Pin To Start button enabled 60](#_Toc367110271)

[Figure 32. The placeholder for the secondary tile 60](#_Toc367110272)

[Figure 33. App bar now showing an Unpin button 62](#_Toc367110273)

[Figure 34. Image of the tile selected for unpinning 62](#_Toc367110274)

[Figure 35. Secondary tile on the Start screen 63](#_Toc367110275)

## Table of Code Samples

[Code Sample 1. Resource dictionaries and theme dictionaries 11](#_Toc367110276)

[Code Sample 2. The Jobs.xml schema 14](#_Toc367110277)

[Code Sample 3. The Equipments.xml schema 15](#_Toc367110278)

[Code Sample 4. Add private method to fetch XML data 18](#_Toc367110279)

[Code Sample 5. Add public method for fetching equipment list for a job 18](#_Toc367110280)

[Code Sample 6. Add public method to fetch equipment details 19](#_Toc367110281)

[Code Sample 7. Bind the equipment list 19](#_Toc367110282)

[Code Sample 8. Bind the equipment details 19](#_Toc367110283)

[Code Sample 9. Overriding default SolidColorBrush resources 22](#_Toc367110284)

[Code Sample 10. Adding new SolidColorBrush and ImageBrush resources 22](#_Toc367110285)

[Code Sample 11. Standard framework TextBlock styles 23](#_Toc367110286)

[Code Sample 12. Overriding framework TextBlock styles 24](#_Toc367110287)

[Code Sample 13. Customize styles in the job item data template 25](#_Toc367110288)

[Code Sample 14. Customize styles in the stock item data template 26](#_Toc367110289)

[Code Sample 15. Customize styles in the job history item template 26](#_Toc367110290)

[Code Sample 16. Handler method activated on search 30](#_Toc367110291)

[Code Sample 17. Search jobs data source method 31](#_Toc367110292)

[Code Sample 18. Add private class members 31](#_Toc367110293)

[Code Sample 19. Load state method for search results page 32](#_Toc367110294)

[Code Sample 20. Add required namespace 32](#_Toc367110295)

[Code Sample 21. Assign the correct template 33](#_Toc367110296)

[Code Sample 22. Change the item container height 33](#_Toc367110297)

[Code Sample 23. Filter check event handler 35](#_Toc367110298)

[Code Sample 24. Item click event 35](#_Toc367110299)

[Code Sample 25. Item click event handler 35](#_Toc367110300)

[Code Sample 26. Define the styles for the parent grid 36](#_Toc367110301)

[Code Sample 27. Applying page header styles 36](#_Toc367110302)

[Code Sample 28. Applying styles to the items control 37](#_Toc367110303)

[Code Sample 29. Add namespace to JobDetails.xaml.cs 39](#_Toc367110304)

[Code Sample 30. Add Handler for DataTransferManager.DataRequested event 39](#_Toc367110305)

[Code Sample 31. Remove Handler for DataTransferManager.DataRequested event 39](#_Toc367110306)

[Code Sample 32. Add DataRequested handler 40](#_Toc367110307)

[Code Sample 33. Create content in HTML format for sharing 41](#_Toc367110308)

[Code Sample 34. Adding an app bar in EquipmentDetails.xaml 43](#_Toc367110309)

[Code Sample 35. Add command button to the app bar 43](#_Toc367110310)

[Code Sample 36. Add event handler for app bar button 44](#_Toc367110311)

[Code Sample 37. Generate XML document for value to be displayed in live tile 45](#_Toc367110312)

[Code Sample 38. Add pending jobs method 46](#_Toc367110313)

[Code Sample 39. Create an XML document for each job 46](#_Toc367110314)

[Code Sample 40. Call the method to create live tiles 46](#_Toc367110315)

[Code Sample 41. Enable tile to queue notifications 47](#_Toc367110316)

[Code Sample 42. Add the Semantic Zoom control 49](#_Toc367110317)

[Code Sample 43. Move GridView inside ZoomedInView 49](#_Toc367110318)

[Code Sample 44. Item template for zoomed-out GridView items 50](#_Toc367110319)

[Code Sample 45. Zoomed-out view 50](#_Toc367110320)

[Code Sample 46. Assign data source to the zoomed-out view 50](#_Toc367110321)

[Code Sample 47. ListView control for a narrow, “snapped” view 53](#_Toc367110322)

[Code Sample 48. Setting up visual states for the page 54](#_Toc367110323)

[Code Sample 49. Register a handler for the Window.SizeChanged event 55](#_Toc367110324)

[Code Sample 50. Method for changing visual states 55](#_Toc367110325)

[Code sample 51. LoadState method with call to UpdateVisualState 55](#_Toc367110326)

[Code Sample 52. Adding the app bar with PinToStart button in JobDetailPage.xaml 57](#_Toc367110327)

[Code Sample 53. Adding the handler for the click event 58](#_Toc367110328)

[Code Sample 54. Add method to check pin state for a given job 58](#_Toc367110329)

[Code Sample 55. Create secondary tile placeholder 58](#_Toc367110330)

[Code Sample 56. Enable toggling of the Pin To Start button 59](#_Toc367110331)

[Code Sample 57. Method for pinning a secondary tile to the Start screen 59](#_Toc367110332)

[Code Sample 58. Event handler for Loaded event of Pin To Start button 60](#_Toc367110333)

[Code Sample 59. Call the unpin method 61](#_Toc367110334)

[Code Sample 60. Unpin a job from the Start screen 61](#_Toc367110335)

[Code Sample 61. Code to replace in the OnLaunched method of App.xaml.cs 63](#_Toc367110336)

[Code Sample 62. Replacement code enabling secondary tile entry 63](#_Toc367110337)

1. Hereafter “App Clinic,” for short. [↑](#footnote-ref-2)