Slide 1: Consumer Preview

The software we're using today is a beta release. My understanding is we are many months from RTM. Microsoft wants us to make it clear that although what we have is really pretty good, there are substantial improvements coming. Don't draw any negative conclusions from what you see today, but secretly they do love it when you talk about the good stuff.

Since this is a beta release, some or all of what I talk about today could change tomorrow. Or not. Sorry if it does, it's out of my control. A LOT of things changed between Developer Preview and Consumer Preview, just for reference.

And as a demo disclaimer, I'm running a beta version of a new OS in a third party virtual machine. I'm using pre-beta development tools, and accessing services on my host machine. This might be a bumpy ride, or it may be the one demo that goes off trouble free.

Slide 2: Win8 Platform and Tools

This slide has been the cause of most of misinformation. The first thing to note is that this is not a heatmap—the size of the box does not represent the relative importance. This is just to illustrate the different technology stacks we can use to develop Windows applications.

Although HTML and JavaScript are represented, I need to reiterate—we are building native Windows applications. The web stacks are a different diagram, and were not the focus of Build Windows.

Slide 3: Why Metro apps?

So why would we build a Metro-style application over a traditional desktop application? First and foremost in my mind is compatibility on both x86/x64 and ARM devices. Metro-style apps will be distributed through the Windows Store, which is huge. Before the application store model, think about what it took to sell an application. Part of distribution via the Windows Store is a trial API, where customers can download a full featured limited trial of your application, then unlock the app when they decide to buy. All of the purchase, licensing, upgrade notifications and in-app purchases are handled for us by the Store's commerce engine.

There is an API for inter-app communication, which is used, for example, if we wanted to allow people to share achievements from within our app. The user no longer has to switch apps or authorize a new one to alert a crushing victory in Angry Birds. Instead, we'd use the hooks to open UI parts from Twitter or Facebook apps used to post a status right inside our app.

WinRT includes optimizations for touch and gestures, and provides support for cloud sync of user data via the Live API. This means a user's settings can roam to any Internet connected device. WinRT also enables Live Tiles.

WinRT also handles media capture and storage, as well as libraries for XML parsing, networking, and access to devices. Like WP7, access to local resources must be declared in the application manifest and the user must grant permissions for certain accesses.

An app can also specify itself as the default handler for domains, so the app will open instead of the browser when the user browses to that domain. Header tags on a web page can also indicate the handler application if it's present.

Slide 4: Two Runtimes

Up until now, we have had one Windows runtime to code against--Win32. In Windows 8, a second runtime was added--WinRT. In the current Microsoft parlance, applications which run in Win32 are called "desktop applications", while those which run in WinRT are called "Metro style apps". The claim "if it runs on Windows 7, it runs on Windows 8" is referring to the Win32 desktop applications and the inclusion of Win32.

With WinRT, Microsoft has created a sandboxed runtime where XAML is the UI structure. We can use either managed (.NET) or unmanaged (C/C++) as the code behind for the XAML. In WinRT, XAML is not a browser plugin like Silverlight, or a standalone library like WPF; XAML is a native part of Windows.

Sandboxing is important. Applications run in their own "application containers" (similar to IIS app pools) and install their dependencies into their own folders. When an app crashes, it won't be able to take down other apps or the OS. There will be duplications of DLLs on your had drive, but each app will have its own requirements in its own folder. Sandboxing also means we have new hoops to jump through to get applications to talk to one another and to the device.

The .NET utilized by WinRT is not a separate version. It's the same set of libraries, but a WinRT profile exposes (or "projects") only the methods and properties allowed in WinRT. You can still get funky with the GAC assemblies, but your app will be rejected from the Windows Store.

There is a second UI technology Microsoft added to WinRT, it's the one that caused all the fuss, and the one we'll be looking at today. We can now build native Windows applications using HTML and JavaScript. This addition was widely seen as a replacement for other technologies, but is actually just the addition of another option for application development. In addition to XAML, we can now use HTML and CSS for our UI, and JavaScript for our code behind. Metro apps built using HTML/JS are run in an HTML Host process, which runs in its own application container.

Architecturally speaking, Metro style apps are very similar in architecture to Windows Phone 7 apps.

Slide 5: Buzzwords

Windows 8 was touted as "a complete reimagining of Windows", which means Microsoft's buzzword team went full-out with new ways to describe Windows. "Clicking buttons" is so Windows 7, in Windows 8, we'll "interact with charms".

Metro-style apps are “tailored” for both the device and the user. Full screen apps immerse the user in the experience, and two apps can be snapped for user multitasking (note that’s user multitasking, not app multitasking). Metro apps are always connected, and the asynchronous programming model ensures the UX is fluid. Apps should be designed to be interactive and touch-first.

Slide 6: Dev tools

Right now there are three tools we can use to build Metro-style applications. VS 11 Express preview ships in the Tools ISO. VS 11 Ultimate preview is available on the USB key Build attendees got with their tablets, and you can also download it separately. Blend 5 is also in the Tools ISO, and can be used to style both XAML and HTML Metro apps. Take note, Blend 5 is not meant for web apps—once you edit HTML in Blend, you’re committed to Metro.

In both Visual Studio 11s, there are new templates for Metro-style apps, and the same templates are available across all languages.

Slide 7: Dev Experience

When developing HTML-based Metro-style applications, we can think of the HTML page as the UI, and the JS file as the code-behind. Writing these apps is more like writing user controls than building websites. We have to put our HTML code in certain blocks on the web page, and put our JavaScript in certain blocks in the JS file for everything to be processed correctly. To accommodate multiple device sizes, layout needs to flow, but that means layout can flow awkwardly, so provide positioning guides beyond just “align” (e.g., CSS grids). Cross-domain requests and unsafe script insertions will cause some calls to fail. Cross domain requests are a jQuery security feature which might need to be overruled; that’s done with a single line of code. For this demo, I couldn’t get the jQuery templating to work because the innerHTML calls are seen as unsafe. That’s not to say it can’t be done, we just need to spend a little more time working on it.

Slide 8: Metro Contexts

When a Metro-style HTML application is running, there are two contexts which control access to resources. The “local context” provides full access to the WinRT, and has limited access to web resources. Scripts can’t be pulled from arbitrary URLs, so they need to be included as local resources. The “web context” provides full access to web resources, but has no access to the WinRT. We can navigate to arbitrary URLs, and can load scripts from CDNs. Depending on the application, the biggest thing we lose in the web context is access to the accelerometer, gyroscope, etc.

Slide 9: Data Access

Not a surprise when you think about it, but this is a significant departure from the past. Microsoft wants to enforce asynchronous communications, so there are no SQL Server drivers. Data-centric apps will rely on WCF services. If WCF isn’t your thing, stay tuned, Microsoft has a couple things in the pipeline to make developing WCF significantly easier.

Local data storage is also an option. For JavaScript apps, we can access an app's isolated storage (this is what we'll use to save state when an app is tombstoned), IndexedDB (via IE 10), and local libraries and devices through WinRT.

Slide 10: Metro App Design

The current definitive reference on designing Metro-style apps is this video from Build. Building the UI is familiar in many ways to good web design. Unlike web design though, there is a greater importance on using the latest capabilities in CSS, HTML and JavaScript to create fluid and interactive layouts. This means using media queries to determine the device's display properties, and applying the correct styles to our layouts.

Do not use absolute positioning or fixed layouts. Microsoft recommends using CSS grids to increase fluidity, and has provided a set of new "flexible" controls (such as ListView and CSS Flexible Box) can rearrange and redistribute their contents automatically.

For graphics, rely on SVG where you can, or provide three pre-sized image resources. By following the correct naming convention, the correct images will be loaded automatically based on 100%, 140% or 180% scaling.

Two apps can be snapped to the screen, either evenly split or main/side style. In addition to scaling to the appropriate resolution, we need to accommodate for several window sizes, too.

Slide 11: Navigation

We can create multiple-page applications, but the recommendation is to stick to a single-page application and use WinJS fragments for browsing to other local pages, or iFrames to browse to external sites. Even when loading content into iFrames, we are still bound by the app's context. A website will open in a new browser window instead of the iFrame of a local context app unless we add the domain to our ApplicationContentUriRules.

We can declare a Metro app to be the default handler for URLs, and web pages can have header tags which hint at the proper handler if it's present. These are probably best used for LOB apps than a public application.

One of the advantages to the single page navigation is that we can maintain state on the client's machine. Depending on the application's architecture, we can work in a disconnected mode, or fill out multi-page forms without making several roundtrips to the server, or keep data in global data objects to pass between fragments. However, if we're going to do this, we need to be aware of the application's lifecycle.

SPAs are not unique to Metro apps. Microsoft shipped an SPA template in their latest MVC 4 bits, and there are a couple of SPA bootstrappers in the community which follow the same pattern. The implementation is different, but the pattern is the same.

Slide 12: Application Lifecycle

When first launching an app, DOMContentLoaded is called after all the DOM objects and scripts have been created, but probably no content or images have been loaded. This is where we register all the handlers (activated, suspending and resuming). DOMContentLoaded is not raised when an app is resumed—only when an app is activated or a page is refreshed.

The activated event is called when a user starts an application fresh. If necessary, we would load any startup data from an app termination here.

Finally, the window.load event is fired after everything is complete. This is not called when an application is resumed, but is called if a page is refreshed.

When an application is switched to the background, the suspending event is called. This is where we save any state data. If system resources run low, it can terminate the application without a chance to save the user data.

When a suspended app is resumed, we'd load the saved data in this event. Several of the templates, such as the Navigation Application, have this stubbed out in the default.js file.

Data can be stored in a local profile, or in a roaming profile

Slide 13: Debugging

Debugging HTML-based Metro-style apps is a bit of a challenge right now. The HTML host process uses a chromeless IE 10 only. There is no option to "View Source", and no F12 developer tools available. This means we have a heavy reliance on the VS debugger and whatever third party tools come along. Right now, it's a lot of breakpoints and variable watches.

When running an app in debug mode, we can choose a simulator target, sort of like the emulator for WP7. However, the simulator is a reflection of your desktop, so you have to be running Windows 8. You just aren't deploying to the local machine.

Demo: WijmoMVC

For this demo, we'll convert part of an MVC app. The main Sales page is built with Wijmo jQuery UI widgets, which we can see are very interactive. The end result we want with have the same look and feel, and retain the interactivity of the original page. This dashboard page is not true MVC, since we use jQuery's ajax to pull data from a service, rather than a controller retrieving data. The rest of this sample app is actual MVC.

Show the code!!

Demo: New Project

For all language choices, the same five templates are available to us. We'll use a navigation application. Just running the application shows there is a specific place where we need to place our content. We can

also see there is no way to actually close the browser; we have to use a windows switcher to go back to VS, then stop debugging. This is a characteristic of Metro-style apps—there is no way to close the application, they are suspended.

In the JavaScript project, defult.html acts like a master page. Other pages we add (such as homepage.html) are structured so that only fragments are pulled from them. There are also a number of new page templates, once our application is created.

Tour of the solutions—where to put scripts and such!

Demo: Adding jQuery and Wijmo

We reference jQuery and Wijmo in default.html, not individual pages. This is true for any third party library—always default.html. We need the following files:

1. Wijmo CSS
2. Theme CSS and images
3. jQuery, jQuery UI, Raphael, and a couple support files
4. Wijmo open and complete

The hooks to WinRT are in the WinJS libraries, which are found in their own folder, and referenced in default.html also.

Demo: HTML and JS

When placing the HTML on the page, we put it in the section with a role=main. Any other place on the page and it won't be included.

In our HTML code, we add two DIVs for the charts, and a grid which will be enhanced. Nothing else is required, since the charts are created in the browser by Raphael.

In the JavaScript file, we put our JavaScript code in the "ready" function. This function is part of the single-page architecture, and is called when the page is set up, similar to the point at when jQuery(document).ready() is called.

In our jQuery, we're making three ajax calls to three services, then passing the data to the proper methods to generate the charts. Although I copied and pasted the code, you can see how simple it is to use Wijmo.

Conclusion

Building Metro-style applications in HTML and JavaScript has a very familiar feel, although we do have to comply with the restrictions of the application design. Overall, the process is very simple, and very powerful applications can be built with the Wijmo jQuery UI widgets.