Fundamentos de WinJS

Alex Casquete

Email: acasquete@outlook.com
LinkedIn: https://www.linkedin.com/in/alexcasquete/

Junio 2012

RESUMEN

Uno de los grandes propósitos que Microsoft se ha marcado con Windows 8 es atraer a los desarrolladores web al mundo de las aplicaciones Metro. A partir de ahora, cualquier desarrollador con conocimientos web puede comenzar a crear aplicaciones nativas para Windows 8 utilizando JavaScript y HTML 5. En este camino no estamos solos, contamos con *Windows Library for JavaScript* (WinJS), una librería que nos facilita la creación de aplicaciones Metro utilizando JavaScript.

WinJS nos proporciona junto con las implementaciones de patrones y métodos para escribir código orientado a objetos, los controles de interfaz de usuario, objetos para enlace de datos, creación de plantillas, control de navegación y animaciones. Y aunque no se trate de JavaScript, también forman parte de esta librería las hojas de estilo que dan a las aplicaciones el estilo Windows Metro. En este artículo daremos un repaso a todos estos aspectos de WinJS que resultan fundamentales para comenzar a desarrollar aplicaciones Metro con JavaScript.

Nota: Este artículo se publicó en el número 93 de la revista dotNetManía en junio de 2012.

ABSTRACT

One of the great purposes that Microsoft has set with Windows 8 is to attract web developers to the world of Metro applications. From now on, any developer with web knowledge can start creating native applications for Windows 8 using JavaScript and HTML 5. In this way, we are not alone; we have the Windows Library for JavaScript (WinJS), a library that facilitates the creation of Metro applications using JavaScript.

WinJS provides us with the implementations of patterns and methods for writing object-oriented code, user interface controls, objects for data binding, template creation, navigation control, and animations. And although it is not JavaScript, the style sheets that give the applications the Windows Metro style are also part of this library. This article will review all these aspects of WinJS that are essential to start developing Metro applications with JavaScript.

Note: This article was published in the issue 93 of June 2012 of the dotNetMania magazine.

1 Comenzando con WinJS

Al crear un nuevo proyecto de aplicación Metro con JavaScript se agrega automáticamente una referencia a todos los ficheros JavaScript que componen **WinJS**, incluyendo las hojas de estilo CSS (Figura 1). Estos ficheros no se copian en nuestro proyecto, sólo son una referencia a los archivos que están físicamente en la carpeta *Program Files*.

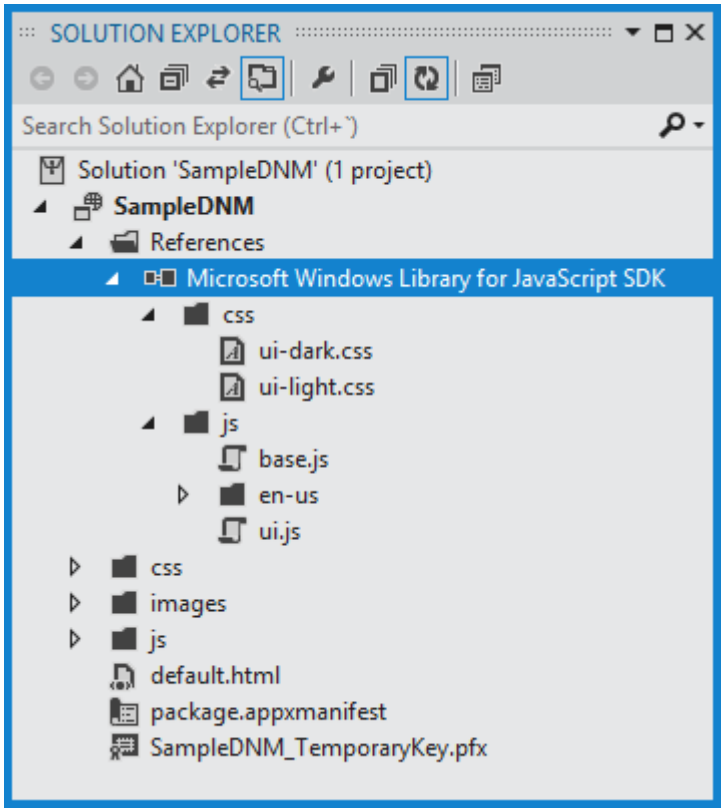

Figura 1 – REFERENCIAS A WINJS EN EL EXPLORADOR DE LA SOLUCIÓN

Además de tener la referencia en el proyecto, para poder hacer uso de **WinJS**, tenemos que referenciar los archivos que necesitemos en cada página HTML de nuestro proyecto. En la Figura 2 se muestran las referencias incluidas automáticamente en la página principal (*default.html*) del cualquier proyecto generado mediante las plantillas de Visual Studio (VS).

```
<!-- WinJS references -->
<link href="//Microsoft.WinJS.0.6/css/ui-dark.css" rel="stylesheet">
<script src="//Microsoft.WinJS.0.6/js/base.js"></script>
<script src="//Microsoft.WinJS.0.6/js/ui.js"></script>
```

Figura 2 – REFERENCIAS A LOS FICHEROS JS Y CSS

La primera referencia es a la hoja de estilo que nos proporciona los estilos base para los elementos HTML y controles **WinJS**. La hoja de estilo oscuro (*ui-dark.css*) –la que se utiliza por defecto– es una de las dos hojas de estilo que tenemos disponibles y está recomendada para aplicaciones que muestren imágenes o video. La otra, la de estilo claro (*ui-light.css*), es el esquema de estilo alternativo aconsejado para aplicaciones que muestren mucho texto. También podemos crear nuestra hoja de estilo personalizada partiendo de una de las dos y sobrescribiendo los estilos que queramos cambiar.

Las otras dos referencias son a dos archivos JavaScript (*base.js* y *ui.js*) que contienen todo el código que nos proveen los controles **WinJS**, los objetos para manipular información y funciones *helper* que nos van a ayudar a escribir código orientado a objetos. Si miramos el contenido de alguno de estos archivos vamos a tener la oportunidad de ver como están organizados y descubrir los patrones utilizados para organizar el código.

2 Patrones JavaScript y WinJS *Helpers*

En JavaScript es muy fácil escribir código que rápidamente se vuelve difícil de mantener. Esta quizás sea la queja más recurrente de los contrarios a JavaScript. Posiblemente esto es debido a que hay algunas características del lenguaje que pueden hacer que escribamos mal código, pero vamos a ver que es el propio lenguaje el que también aporta algunas soluciones a esos problemas.

Sin duda, el primer gran problema de JavaScript es el ámbito de las variables. Muchos lenguajes (como C# o C++) tienen un ámbito de bloque (*block scope*), en los que se pueden definir variables que sólo afecten al bloque de código entre llaves. Sin embargo, JavaScript no tiene está característica, sólo podemos definir un ámbito global o de función. Esto supone un problema cuando las aplicaciones crecen en complejidad, porque a mayor número de variables globales más difícil se hace la depuración.

En aplicaciones web, donde estamos acostumbrados a utilizar JavaScript, tendemos a no preocupamos por el ámbito de las variables porque cuando navegamos a otra página todo el DOM se limpia de memoria. Sin embargo, cómo veremos más adelante, en las aplicaciones Metro no se navega entre páginas sino que todo sucede en una sola página y por lo tanto debemos ser cuidadosos y limitar al máximo el número de esas variables globales. Una forma de conseguirlo es mediante el patrón Módulo (*Module Pattern*), que básicamente consiste en englobar el código en una función anónima autoejecutada (Figura 3), consiguiendo que las variables que tienen ámbito global pasen a tener ámbito de función. Además, y aunque en JavaScript no existe el concepto de espacios de nombres, podemos utilizar los objetos para simularlos.

```
var moduleName = (function(){

  var privateVariable = 1;
  var privateMethod = function(){
    console.log(privateVariable);
  };

  return {
    publicVariable: "Value",

    publicMethod: function(){
      privateVariable++;
      privateMethod();
    }
  };

})();

moduleName.publicMethod();
```

Figura 3 – IMPLEMENTACIÓN BÁSICA DEL PATRÓN MÓDULO

WinJS nos proporciona los métodos *helper* para poder definir espacios de nombres fácilmente (Figura 4). El método **WinJS.Namespace.define** requiere que pasemos el nombre del espacio de nombres y los miembros. Un espacio de nombres puede contener clases, funciones, constantes y otros espacios de nombres. Además, con **WinJS** la definición de un espacio de nombres es un proceso aditivo, es decir, si definimos el mismo espacio de nombres en ficheros distintos estaremos ampliando su funcionalidad, no sobrescribiéndola.

```
WinJS.Namespace.define("moduleName", {

    _privateVariable: 1,
    _privateMethod: function () {
        console.log(this._privateVariable);
    },

    publicMethod: function () {
        this._privateVariable++;
        this._privateMethod();
    }
});

moduleName.publicMethod();
```

Figura 4 – EJEMPLO DEFINICIÓN NAMESPACE CON WINJS Y LLAMADA A UN MÉTODO

De forma similar a cómo definimos los espacios de nombres, podemos definir objetos utilizando el método ***define*** de **WinJS.Class** pasando el constructor y los miembros de la instancia.

```
var myClass = WinJS.Class.define(
    function () {
        // Constructor
    },
    {
        myMethod: function () { }
    });

var obj = new myClass();
obj.myMethod();
```

Figura 5 – DEFINICIÓN E INSTANCIACIÓN DE UN OBJETO CON WINJS

Mediante estos métodos vamos a poder implementar las características de OOP fácilmente en nuestras aplicaciones Metro con JavaScript, a la vez que obtenemos un código reusable y bien organizado.

3 Programación asíncrona

Otro patrón incluido en la librería de **WinJS** tiene el objetivo de facilitar la implementación y sincronización de procesos asíncronos. Aunque en JavaScript podemos lanzar procesos asíncronos mediante los controladores de eventos y la función *setTimeout* (Figura 6), estos tienen el inconveniente de que el código tiende a hacerse más complejo cuando añadimos gestión de errores o tenemos que sincronizar varias llamadas asíncronas, por ejemplo, cuando hacemos una llamada a un servicio y a partir de la respuesta de éste realizamos una nueva llamada. La comunidad JavaScript propuso

una solución a este problema a través del estándar **Common JS Promises/A** que define una *Promise* o compromiso.

```
// Controlador de eventos
var body = document.querySelector("body");
body.addEventHandler("load", function () { … }, false);

// setTimeout
window.setTimeout(function () { … }, 1000);
```

Figura 6 – EJEMPLO SETTIMEOUT y ATTACHED EVENT HANDLER

Básicamente una *Promise* es un objeto que nos «promete» un resultado en algún momento futuro y nos permite programar cualquier operación cuando ese valor esté establecido. Esto nos permite escribir código no bloqueante que se ejecuta de forma asíncrona y sin necesidad de escribir código para manejar la sincronización.

La especificación es muy sencilla, y dice simplemente que hay que tener un objeto con un método llamado **then** y que este método debe aceptar tres parámetros. Estos tres parámetros serán tres funciones *callback* que se llamarán cuando la *Promise* se complete, cuando se produzca un error o cuando se produzca algún evento de progreso. En **WinJS** el objeto que se utiliza es **WinJS.Promise** y se devuelve en todos los métodos asíncronos.

```
WinJS.xhr({ url: "http://feeds.feedburner.com/DNMPlus" }).then(
  function (result) {
      console.log("Promise completada con éxito.");
  },
  function (error) {
      console.log("Promise completada con error.");
  },
  function (progress) {
      console.log("Promise en progreso.");
  });
```

Figura 7 - USO DEL OBJETO PROMISE DEVUELTO POR EL MÉTODO XHR

En la Figura 7 vemos el ejemplo de una llamada al método **WinJS.xhr** que realiza una petición **XmlHttpRequest** y devuelve un objeto *Promise*. Cuando se reciba una respuesta correcta del servidor, la *Promise* se completará y se invocará la función que se pasa como primer parámetro del método **then**. Si, por el contrario, el servidor devuelve un error, la *Promise* se completará, pero en ese caso la función que se invocará será la que se pasa en el segundo parámetro. El método **WinJS.xhr** notifica los tres estados, pero esto no sucede en todos los métodos que devuelven *Promises*, ya que es su implementación es opcional.

Otro aspecto importante y que **WinJS** no descuida es el referente a la coordinación de varias *Promises* **(**Figura 8**)**. Disponemos de los métodos **any** y **join** que permiten definir *Promises* compuestas, aceptan como parámetro un *array* de *Promises* y a su vez devuelven una *Promise* que se completa cuando alguna se completa (método *any*), o cuando todas se completan (método *join*).

```
promises.push(WinJS.xhr({ url: "http://domain.org/file1" }));
promises.push(WinJS.xhr({ url: "http://domain.org/file2" }));

WinJS.Promise.join(promises).then(function (results) {
    console.log("Todas las Promises se han completado");
});

WinJS.Promise.any(promises).then(function (results) {
    console.log("La Promise " + results.key + " se ha completado");
});
```

Figura 8 – USO DE LOS MÉTODOS ANY Y JOIN PARA COORDINAR PROMISES

Con el uso de *Promises* disponemos del mecanismo perfecto para la gestión de interacciones de las API asíncronas como Windows Runtime. Podemos utilizarlas para realizar cualquier operación asíncrona, como puede ser guardar o leer un archivo, obtener una imagen o video de la webcam o descargar un archivo.

4 Controles WinJS

Pasamos ahora a ver otra función esencial de **WinJS** que no es otra que la de proveernos de todos los controles básicos diseñados para las aplicaciones Metro (Figura 9). Los controles **WinJS**, a diferencia de los controles clásicos HTML, no tienen un elemento dedicado, es decir, si queremos añadir la barra de aplicación (*App Bar*) no lo hacemos añadiendo el elemento <appbar /> a nuestro código HTML, sino que crearemos un objeto JavaScript enlazado a un elemento HTML.

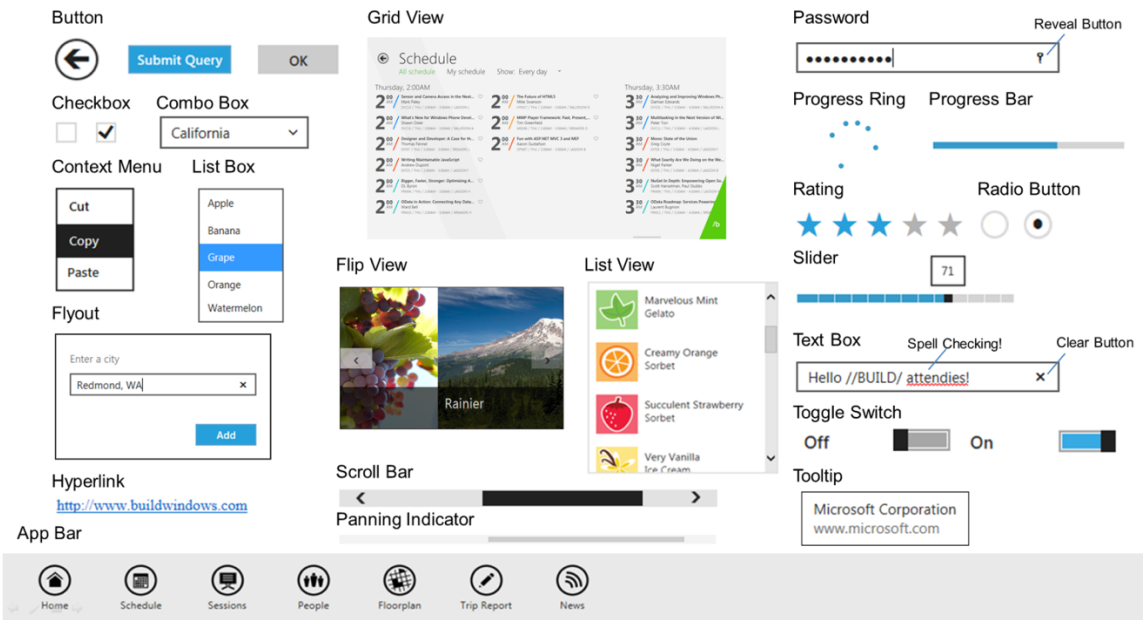

Figura 9 – CONTROLES WINJS PARA APLICACIONES METRO

En la Figura 10 vemos como se define el control **AppBar** a partir de un elemento div y utilizando el atributo personalizado **data-win-control** especificamos el tipo de control que queremos mediante el nombre cualificado. Con esto estamos indicando que el elemento será el *host* del control **WinJS**.

```
<div id="appbar" data-win-control="WinJS.UI.AppBar" data-win-
options="{sticky:true}">
   <button data-win-control="WinJS.UI.AppBarCommand" data-win-
options="{id:'cmd', label:'Command', icon:'placeholder'}"></button>
</div>
```

Figura 10 – DECLARACIÓN DE UN OBJETO APPBAR Y APPBARCOMMAND

Una vez lo tenemos declarado en el HTML tenemos que llamar al método **WinJS.UI.processAll** que se encarga de recorrer todo el código HTML e instanciar los controles **WinJS** que contiene el documento. Si estamos utilizando alguna plantilla de VS, está llamada se realiza de forma predeterminada en el fichero *default.js*.

Además de crear el objeto, podemos establecer el valor de las propiedades de los controles **WinJS** mediante el atributo **data-win-options**, especificando una cadena que contenga los pares propiedad/valor. En el ejemplo se establece la propiedad *sticky* de la barra de aplicación y las propiedades *id*, icon y *label* del control **AppBarCommand**.

Los controles **WinJS** se pueden crear declarativamente, como hemos visto, o programáticamente llamando al constructor del control desde el código JavaScript. Por ejemplo, si queremos instanciar un control **DatePicker** utilizaremos el siguiente código (donde *target* es el elemento HTML que hará de host del control WinJS): *var dateControl = new WinJS.UI.DatePicker(target, { maxYear: 2020 });*

5 DataBinding

WinJS incluye un motor de *databinding* que nos va a permitir relacionar la interfaz de usuario con un conjunto de datos que tenemos en el código JavaScript, de forma similar a como lo podemos hacer con *Knockout.js* u otras librerías similares.

El atributo **data-win-bind** nos permite asociar un atributo de un elemento HTML con el valor de una propiedad de un objeto JavaScript, así evitamos el tener que acceder desde nuestro código a cada uno de los controles HTML.

default.html

```
<div id="contactTemplate">
    <h1 data-win-bind="innerText:name"></h1>
    <p data-win-bind="innerText:address"></p>
    <p data-win-bind="innerText:phone"></p>
</div>
```

default.js

```
var contact = {
    name: "José García",
    address: "Gran vía, 536",
    phone: "555-0143"
};

WinJS.Binding.processAll(document.body, contact);
```

Figura 11 – DATABINDING DE UN SOLO ELEMENTO

En la Figura 11 tenemos tres atributos, uno para cada propiedad del contacto. Debemos usar **data-win-bind** para establecer las propiedades del elemento HTML asociado con los datos. Es importante destacar que no necesitamos establecer el id de ningún elemento ya que desde el código JavaScript no vamos a necesitar acceder a ellos.

La parte izquierda del valor atributo **data-win-bind** indica la propiedad de destino (*innerText*) y la parte de la derecha indica el origen de datos. Y de forma análoga a lo que hacíamos para procesar todos los controles, para que se procesen todos los *bindings* tenemos que llamar a la función **WinJS.Binding.processAll**, así que tenemos que añadir la llamada después de que se procesen todos los controles.

Esta función, **WinJS.Binding.processAll**, permite que le pasemos el elemento inicial por el que se empezará a buscar los elementos que se tienen que enlazar y un objeto **DataContext**, que servirá como origen de datos. En el ejemplo se ha pasado *document.body* para que procese todos los elementos del cuerpo del documento y el objeto *dataContext* que contiene los datos del contacto.

Si queremos habilitar las notificaciones a la interfaz de usuario al hacer un cambio en el origen de datos, debemos utilizar la función **WinJS.Binding.as** para crear un *proxy* observable del objeto, esto significa que cualquier cambio que hagamos en el objeto JavaScript se verá reflejando en los elementos que estén enlazados a ese objeto. Esta notificación sólo sucede en un sentido (*One way*), del objeto al control. Si modificamos el valor en la interfaz de usuario, el objeto JavaScript no actualizará su valor.

6 Template binding

Imaginemos que en lugar de mostrar un solo elemento queremos mostrar una lista de contactos utilizando el mismo formato. Mediante las plantillas **WinJS** podemos especificar un control que podemos reutilizar. En la Figura 12 podemos ver cómo se declara una plantilla. La única diferencia con el código anterior es que hemos asignado el atributo **data-win-control** para definir un control **WinJS.Binding.Template** y hemos creado otro elemento div (*contactContainer*) que servirá de contenedor para mostrar el resultado una vez se haya procesado la plantilla.

default.html

```
<div id="contactTemplate" data-win-control="WinJS.Binding.Template">
    <h1 data-win-bind="innerText:name"></h1>
    <p data-win-bind="innerText:address"></p>
    <p data-win-bind="innerText:phone"></p>
</div>

<div id="contactContainer"></div>
```

default.js

```
var contact = WinJS.Binding.as([
    { name: "José", address: "Gran vía, 536", phone: "555-0143" },
    { name: "Carlos", address: "Avda. Andalucía, 81", phone: "555-0144" },
    { name: "Alberto", address: "Ctra. Málaga, 32", phone: "555-0145" }
]);

var template = new WinJS.Binding.Template(document.getElementById("contactTemplate"));
var container = document.getElementById("contactContainer");

for (var i = 0; i < contact.length; i++) {
    template.render(contact[i], container);
}
```

Figura 12 – DECLARACIÓN DE UNA PLANTILLA ENLAZADA A UN OBJETO OBSERVABLE

En el código, convertimos el elemento *contactTemplate* a una plantilla **WinJS** mediante el objeto **WinJS.Binding.Template** y la dibujamos en el elemento contenedor mediante el método **render**. En este código no vemos la llamada al método **WinJS.Binding.processAll** ya que el método **render** realiza el enlace a datos automáticamente.

Para terminar, podríamos separar más el código pasando la definición de la plantilla (el elemento *contactTemplate)* a un fichero independiente. Si hacemos esto tendremos que cambiar los parámetros que se pasan en el constructor del objeto **WinJS.Binding.Template**.

var productTemplate = new WinJS.Binding.Template(null, { href: "/templates/productTemplate.html" });

7 Navegación

Para terminar este repaso a las funcionalidades básicas de **WinJS**, vamos a ver las diferencias del modelo de navegación de las aplicaciones Metro con JavaScript respecto a la web tradicional. La forma más sencilla que tenemos de movernos de una página a otra en las aplicaciones web clásicas es a través un enlace. Este modelo es conocido como navegación multipágina, en la que cada página carga de nuevo todo el código HTML, JavaScript y CSS. Sin embargo, este modelo necesita que tengamos que serializar el estado para pasarlo de una página a otra.

Las aplicaciones Metro con JavaScript, como hemos visto antes al hablar de los patrones, utilizan una navegación de página única. Con este modelo tenemos una gestión de

estado de la aplicación mucho más sencilla, pero por el contrario añadimos dificultad al tener que diseñar toda la aplicación para que se ejecute en un solo fichero HTML. Esto significa que podemos seguir dividiendo la aplicación en varios ficheros, pero que en lugar de navegar de página en página, la aplicación debe cargar los documentos en la página principal.

Para facilitarnos las cosas un poco, las plantillas de aplicación *Grid*, *Split* y *Navigation* que trae VS contienen un control de navegación llamado **PageControlNavigator** definido en el fichero *navigator.js* y declarado en la página principal (default.html), que nos proporciona toda la infraestructura para poder implementar la navegación entre páginas. Este control lo que hace es procesar la página a cargar mediante la función **WinJS.UI.Pages.render** y añadir el código HTML resultante en el elemento div *contentHost* de la página principal.

En **WinJS**, toda la funcionalidad de navegación se encuentra en el espacio de nombre **WinJS.Navigation** que expone los métodos y propiedades para realizar la navegación entre páginas y movernos por el historial. Si queremos navegar a otra página, en lugar de utilizar el comportamiento predeterminado de un enlace, tenemos que llamar a la función **WinJS.Navigation.navigate** pasando la URL del fichero a cargar. Al hacer esto la aplicación guarda el historial de navegación y podemos ir hacia atrás o hacia delante mediante las funciones **WinJS.Navigation.back** y **WinJS.Navigation.forward**.

```
document.getElementById("btnNavigate").addEventListener("click", function (e)
{
            WinJS.Navigation.navigate("/html/Page2.html");
        });
```

Figura 13 – CÓDIGO PARA NAVEGAR A OTRA PÁGINA AL PULSAR UN BOTÓN

Al utilizar estos métodos de navegación obtendremos una experiencia mucho más cercana a la de una aplicación cliente y nos despreocupamos de la gestión de estado de la aplicación.

8 Conclusiones

En este artículo hemos realizado una primera aproximación a las principales funcionalidades que nos proporciona WinJS para la programación de aplicaciones Metro con JavaScript. Primero, hemos visto que siguiendo una serie de patrones podemos obtener un código reusable y organizado para hacerlo más mantenible. Además, gracias a los métodos *helper* podemos implementar OOP fácilmente, definiendo espacios de nombres y clases. En el caso de la programación asíncrona, el patrón *Promise* facilita la sincronización de varios procesos asíncronos. Hemos visto también como se declaran los objetos WinJS y cómo podemos separar responsabilidades utilizando *databinding* y las plantillas. Por último, hemos revisado el funcionamiento del modelo de navegación de página única y las ventajas que tenemos al utilizar las plantillas de proyecto de VS.

Referencias

Microsoft Docs. WinJS. http://msdn.microsoft.com/en-us/library/windows/apps/br211377.aspx

CommonJS. Promises/A. http://wiki.commonjs.org/wiki/Promises/A