# Preparativos

* Abrir la consola con IISExpress y dejar listo para levantar el /siteid:2
* Limpiar el cache del navegador.
* (Opcional) Cambiar el puerto en el IISExpress y el VS del BakeryTemplate
* Abrir WebMatrix y mostrar la aplicación de Bakery.
* Limpiar los “Watch Expressions” del navegador

# Definición

Es una forma arquitectónica para aplicaciones web. De tal manera que las diseñamos e implementamos para que estas se ejecuten persistentemente sobre la misma página sin necesidad de recargar.

Nosotros sabemos lo buenas que son las aplicaciones web (desplegarse en un solo lugar, accederlas en múltiples dispositivos, etc) pero las aplicaciones de escritorio también son buenas (son bastante responsivas y pueden trabajar offline).

Este tipo de aplicaciones combinan los beneficios de ambas aplicaciones, es decir no so solo tienen todos los beneficios de las aplicaciones web, sino también son responsivas, ricas y pueden trabajar offline. Estas aplicaciones son generalmente implementadas utilizando HTML5 y Javascript.

# Ejemplos

* Trello es una herramienta online de colaboración que permite nos permite organizar cualquier tipo de proyecto en paneles. Ejm: Proyectos de software, actividades personales o la casa etc. Puedes agregar tareas, tags, asignar personas a tareas, documentos y un sinfín de cosas sin necesidad de estar recargando la aplicación una y otra vez.
* Hipmunk es un buscador que nos permite buscar vuelos aéreos, pone un origen y un destino y te muestra todas las alternativas, te muestra esa ventana, puedes filtrar, cambiar los criterios, comprar. Responde de inmediato xq no necesita ir hasta el servidor para realizar todas esas actividades y simplemente las realiza en el cliente.
* Gmail App, parece una aplicación nativa, que puede ser descarga del appstore como cualquier otra aplicación, cualquier usuario normal puede pensar que es una aplicación nativa, pero esta completamente hecha con tecnologías web y puede incluso trabajar offline.

# Beneficios

* Buena experiencia de usuario: Cuando hablamos de este punto nos referimos a 2 cosas: aplicaciones de interfaz rica, dinámicas y también a la velocidad ya que la respuesta es inmediata muy similar a las aplicaciones nativas.
* Reducen la carga en el servidor: Gran parte del procesamiento de estas páginas se realiza en el cliente, con lo cuál se reduce el trabajo en el servidor, permitiendo que este escale más fácilmente.
* Accesibles a través de cualquier dispositivo: esta es una característica innata de las aplicacaciones web, pero que ahora está tomando nuevamente más fuerza por el auje de los dispositivos móviles. Ya que nosotros podemos evitar pelear con las APIS muy particular de cada mobile y realizar una aplicación web orientada que pueda ser accedido independientemente del dispositivo.
* Similares a las aplicaciones nativas. No solo podemos lograr que estas aplicaciones puedan ser accedidas en diferentes disposivos, sino que estas sean tan dinámicas responsivas y se vean exactamente igual que una aplicación nativa, que un usuario normal no puede notar la diferencia.
* Pueden trabajar offline. De manera similar a las aplicaciones nativas, se puede hacer que estas trabajen completamente desconectadas de la red.
* Pueden desplegarse en App-Stores. Puedes crear una aplicación web que pueda ser empaquetada y desplegada en los appstores de tal manera que un usuario pueda descargarla, instalarla como si fuera cualquier una aplicación nativa.

# DEMO

Abrir la aplicación de Web Matrix e indicar que nuestro objetivo es convertir esa aplicación a una SPA.

# Data

En una página web normal, nosotros necesitamos algún tipo de endpoint que nos sirva las páginas web, css y el js, para que estos sean renderizados en el navegador. Una aplicación estándar, nada nuevo.

Pero en una SPA, si bien es cierto la navegación, generación del HTML se realiza en el cliente. Este cliente necesita que alguien le provea los datos para que sean procesados. Es por eso que necesitamos endpoints que nos provean datos en el servidor, estos datos pueden estar en formato JSON o XML. Usualmente estos servicios son creados y expuestos a través de una API Rest (principios arquitectónicos por los cuales podemos diseñan servicios web basados en el protocolo HTML, debido las ventajas de crear servicios utilizando este enfoque, está desplazando rápidamente a SOAP)

En este caso vamos a utilizar ASP.NET Web API pero podría ser Spring, Sinatra o cualquier otra tecnología web o framework web.

Ya he creado un proyecto ASP.NET y los servicios, ya que nos vamos a enfocar principalmente en HTML 5 y Javascript por lo tanto no pretendo aburrirlos con C# y ASP.NET.

ODATA: Es un protocolo basado en HTTP y Atom que sigue los principios REST. El objetivo principal de OData es ofrecer una forma estándar de consumirdatos a través de la red y conseguir que los consumidores de los servicios hagan uso de una serie de convenciones. Por ejemplo alguna de estas convenciones nos permite realizar queries directamente a los recursos expuestos.

http://localhost:11763/dataservice/products?$filter=id eq 1

http://localhost:11763/dataservice/products?$orderby=Name

Está diseñado para compartir datos sobre todo en entornos públicos e interoperables.

# Organización

## User Interface Patterns

Para organizar y estructura nuestro código en el front-end existen 3 patrones muy comunes: MVC, MVP, MVVM. La intensión no es confundirlos con todas esas flechitas y profundizar en el tema, sino comprender a nivel básico el funcionamiento y diferencia entre estos, ya que sabiendo esto podremos aplicarlos y utilizarlos correctamente.

Seguramente nosotros ya hemos trabajado de alguna manera con un patrón MVC. Este patrón al igual que los demás pretende dividir la aplicación en 3 partes: M, V y un componente intermedio que se encarga de coordinar la interacción entre el modelo y la vista, y gestionar el flujo de la aplicación.

Si recordamos un poco los q hemos trabajado con MVC, el funcionamiento de este patrón consistía en:

* La vista tiene una referencia al modelo, es decir sabe cuál es el modelo que se va a pintar.
* El controller es el encargado de determinar cuál es la vista a mostrar y pasarle un modelo a esta vista.
* La vista no conoce cuál es el controller.

MVP:

* La vista no tiene ninguna referencia del modelo ni tampoco el presenter le pasa un modelo.
* La vista expone diversos setters para que a través de estos el presenter se encargue de actualizar los datos del modelo.
* La vista mantiene una referencia del Presentar y también de la vista.

MVVM

* La vista conoce tanto el modelo como el viewmodel que está utilizando.
* Existe una sincronización en ambos sentidos entre la vista y el modelo.
* Los viewmodels son representaciones en código de las vistas.

## Implementaciones Comunes

En el caso de JS que es lo que a nosotros nos interesa, las implementaciones más comunes son la de MVC y la de MVVM.

## MV\* Patterns

El patrón MVC es el más antiguo dentro de JS y posee múltiples implementaciones, estas son solo algunas de ellas, de estas probablemente la más antigua y conocida sea backbone.js, que nos provee todo lo necesario para realizar aplicaciones web utilizando este patrón.

El patrón MVVM recién hace un par de años ha empezado a surgir fuertemente en Javascript.

Ambos patrones permiten realizar aplicaciones de todo tamaño. Aunque talvez las implementaciones MVC sea un poco más complejas para cosas pequeñas, y las implementaciones MVVM requieran un poco más de trabajo de parte de nosotros para cosas complejas.

Para nuestro ejemplo vamos a utilizar MVVM:

## Knockout.JS

En específico knockoutjs, no xq porque sea mejor sino por motivos particulares a esta sesión:

* Los patrones MVVM en JS son los nuevos hoties dentro de JS y asimismo nosotros seguramente ya conocemos como trabaja MVC y sería bueno explorar este nuevo patrón.
* En otro día del evento ya se habló sobre MVC.

Knockout al ser un patrón MVVM permite que el viewmodel y la vista se mantengan sincronizados en todo momento y cualquier cambio de uno de ellos se vea reflejado automáticamente en el otro. A esta característica la llama Dependency Tracking.

Knockout logra conectar el viewmodel y la vista, es decir logra crear bindings entres estos dos, utilizando los atributos data-bind dentro de HTML.

Pequeño ejemplo utilizando <http://jsfiddle.net/> (Editor en línea para crear snippets de código orientado a lo que es la web)

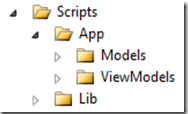
<input data-bind="value:firstname"/>  
<p data-bind="text:firstname"></p>​

var viewModel=function(){  
    firstname=ko.observable("Juan");  
}  
  
ko.applyBindings(new viewModel());​

## Código

Agregar la referencia a Knockout.

Crear la estructura de directorios.



## Pantalla Número 1

**ChooseProductViewModel.js**

var ChooseProductViewModel = function () {  
    var self = this;  
   
    self.products = ko.observableArray();  
   
    self.goToPlaceOrder = function (product) {  
        //Ir la vista PendingOrder  
    };  
   
    self.init = function () {  
        ProductsDataSource.getAll(function (data) {  
            self.products(data);  
        });  
    };  
   
    self.init();  
};

**ProductsDataSource.js**

var ProductsDataSource = (function () {  
    return {  
        getAll: function (success) {  
            $.ajax({  
                type: "GET",  
                url: "/dataservice/products",  
                contentType: "application/json;charset=utf-8",  
                dataType: "json",  
                success: success  
            });  
        }  
    };  
} ());

**View 1**

<div>  
    <!-- For each Product -->  
    <ul id="products">  
        <li class="product">  
            <div class="productInfo">  
                <h3></h3>  
                <img class="product-image"/>  
                <p class="description"></p>  
            </div>  
            <div>  
                <p class="price"></p>  
                <a class="order-button">Order Now</a>  
            </div>  
        </li>  
    </ul>  
</div>

**View2**

<div>  
    <ul id="products" data-bind="foreach: products">  
        <li class="product">  
            <div class="productInfo">  
                <h3 data-bind="text: name">  
                </h3>  
                <img class="product-image" data-bind="attr:{ src:'/content/images/products/thumbnails/' + imagename, alt:name }" />  
                <p class="description" data-bind="text: description">  
                </p>  
            </div>  
            <div>  
                <p class="price" data-bind="text: price">  
                </p>  
                <a class="order-button" data-bind="attr:{ title:name }, click:$parent.goToPlaceOrder">  
                    Order Now</a>  
            </div>  
        </li>  
    </ul>  
</div>

<script type="text/javascript">  
    $(function () {  
        initializeApplication();  
    });  
</script>

**Application**

function initializeApplication() {  
    initializeViewModels();  
}  
   
function initializeViewModels() {  
    ko.applyBindings(new ChooseProductViewModel());  
}

**Referencias**

<script src="~/Scripts/App/Data/productsdatasource.js" type="text/javascript"></script>  
<script src="~/Scripts/App/ViewModels/chooseproductviewmodel.js" type="text/javascript"></script>  
<script src="~/Scripts/application.js" type="text/javascript"></script>

## Pantalla 2

**Application**

function initializeAjaxDefaultParameters() {  
    $.ajaxSetup({  
        dataType: "json",  
        contentType: "application/json;charset=utf-8"  
    });  
}

**PlaceOrderViewModel**

var PlaceOrderViewModel = function (productId) {  
    var self = this;  
   
    self.order = ko.observable();  
   
    self.postOrder = function () {  
        OrdersDataSource.create(self.order, function () {  
            //Ir a la pantalla de confirmación

//Mostrar una alerta de que la orden se creo.  
        });  
    };  
   
    self.init = function () {  
        ProductsDataSource.get(productId, function (product) {  
            self.order(new Order(product));  
        });  
    };  
   
    self.init();  
};

**Models.js**

var Order = function (product) {  
    var self = this;  
    self.address = ko.observable();  
    self.email = ko.observable();  
    self.quantity = ko.observable(1);  
    self.productId = product.id;  
    self.product = product;  
    self.total = ko.computed(function () {  
        return (self.product.price \* self.quantity()).toFixed(2);  
    });  
}

**ProductsDataSource**

var ProductsDataSource = (function () {  
    return {  
        getAll: function (success) {  
            $.ajax({  
                type: "GET",  
                url: "/dataservice/products",  
                success: success  
            });  
        },  
   
        get: function (id, success) {  
            $.ajax({  
                type: "GET",  
                url: "/dataservice/products/" + id,  
                success: success  
            });  
        }  
    };  
} ());

**OrdersDataSource**

var OrdersDataSource = (function () {  
    return {  
        create: function (order, success) {  
            $.ajax({  
                type: "POST",  
                url: "/dataservice/orders",  
                data: ko.toJSON(order),  
                success: success  
            });  
        }  
    };  
} ());

**DeliveryViewModel**

var DeliveryViewModel = function() {  
    var self = this;  
   
    self.chooseProduct = ko.observable();  
    self.placeOrder = ko.observable();  
   
    self.showChooseProduct = function() {  
        self.chooseProduct(new ChooseProductViewModel(self));  
        self.placeOrder(null);  
    };  
   
    self.showPlaceOrder = function(productId) {  
        self.chooseProduct(null);  
        self.placeOrder(new PlaceOrderViewModel(productId,self));  
    };  
   
    self.showChooseProduct();  
};

**Application**

function initializeApplication() {  
    initializeAjaxDefaultParameters();  
    initializeViewModels();  
}

function initializeViewModels() {  
    ko.applyBindings(new DeliveryViewModel());  
}

**ChooseProductViewModel**

var ChooseProductViewModel = function (parent) {  
    var self = this;  
   
    self.products = ko.observableArray();  
   
    self.goToPlaceOrder = function (product) {  
        parent.showPlaceOrder(product.id);  
    };  
   
    self.init = function () {  
        ProductsDataSource.getAll(function (data) {  
            self.products(data);  
        });  
    };  
   
    self.init();  
};

**ChooseProductView**

<div data-bind="with: chooseProduct">  
  
</div>

**PlaceOrderView**

<div data-bind="with: placeOrder">  
    <!-- ko with: order -->  
    <h1 data-bind="text: 'Place Your Order: '+product.name">  
    </h1>  
    <form action="" method="post">  
        <fieldset class="no-legend">  
            <legend>Place Your Order</legend>  
            <img class="product-image order-image" data-bind="attr:{ src:'/content/images/products/thumbnails/' + product.imagename, alt:product.name }" />  
            <ol>  
                <li>  
                    <label for="orderEmail">Your Email Address</label>  
                    <input type="text" id="orderEmail" name="orderEmail" data-bind="value:email" />  
                </li>  
                <li>  
                    <label for="orderShipping">Shipping Address</label>  
                    <textarea rows="4" cols="20" id="orderShipping" name="orderShipping" data-bind="value:address"></textarea>  
                </li>  
                <li class="quantity">  
                    <label for="orderQty">Quantity</label>  
                    <input type="text" id="quantity" name="quantity" data-bind="value:quantity, valueUpdate: 'afterkeydown'" />  
                    x <span id="orderPrice" data-bind="text:product.price"></span>  
                    = <span id="orderTotal" data-bind="text:total()"></span></li>  
            </ol>  
            <p>  
                <input type="submit" value="Place Order" data-bind="click:$parent.postOrder" />  
            </p>  
        </fieldset>  
    </form>  
    <!-- /ko -->  
</div>

**Referencias**

<script src="~/Scripts/App/Models/models.js" type="text/javascript"></script>  
<script src="~/Scripts/App/Data/productsdatasource.js" type="text/javascript"></script>  
<script src="~/Scripts/App/Data/ordersdatasource.js" type="text/javascript"></script>  
<script src="~/Scripts/App/ViewModels/chooseproductviewmodel.js" type="text/javascript"></script>  
<script src="~/Scripts/App/ViewModels/placeorderviewmodel.js" type="text/javascript"></script>  
<script src="~/Scripts/App/ViewModels/deliveryviewmodel.js" type="text/javascript"></script>  
<script src="~/Scripts/application.js" type="text/javascript"></script>

## Pantalla 3

**ConfirmationViewModel**

var ConfirmationViewModel = function (parent) {  
    var self = this;  
    self.goToProducts = function () {  
        parent.showChooseProduct();  
    };  
};

**PlaceOrderViewModel**

var PlaceOrderViewModel = function (productId, parent) {  
    var self = this;  
   
    self.order = ko.observable();  
   
    self.postOrder = function () {  
        OrdersDataSource.create(self.order, function () {  
            parent.showConfirmation();  
        });  
    };  
   
    self.init = function () {  
        ProductsDataSource.get(productId, function (product) {  
            self.order(new Order(product));  
        });  
    };  
   
    self.init();  
};

**DeliveryViewModel**

var DeliveryViewModel = function () {  
    var self = this;  
   
    self.chooseProduct = ko.observable();  
    self.placeOrder = ko.observable();  
    self.confirmation = ko.observable();  
   
    self.showChooseProduct = function () {  
        self.chooseProduct(new ChooseProductViewModel(self));  
        self.placeOrder(null);  
        self.confirmation(null);  
    };  
   
    self.showPlaceOrder = function (productId) {  
        self.chooseProduct(null);  
        self.placeOrder(new PlaceOrderViewModel(productId, self));  
        self.confirmation(null);  
    };  
   
    self.showConfirmation = function () {  
        self.chooseProduct(null);  
        self.placeOrder(null);  
        self.confirmation(new ConfirmationViewModel(self));  
    };  
   
    self.showChooseProduct();  
}

**ConfirmationView**

<div data-bind="with: confirmation">  
    <h1>Order Confirmation</h1>  
    <div id="order-success" class="message" >  
        <h2>Thank you for your order!</h2>  
        <p>  
            We are processing your order and have sent a confirmation email. Thank you for your  
            business and enjoy!  
        </p>  
    </div>  
    <p>  
        <a class="back" data-bind="click:goToProducts">Continue Shopping</a>  
    </p>  
</div>

**Referencias**

<script src="~/Scripts/App/Models/models.js" type="text/javascript"></script>  
<script src="~/Scripts/App/Data/productsdatasource.js" type="text/javascript"></script>  
<script src="~/Scripts/App/Data/ordersdatasource.js" type="text/javascript"></script>  
<script src="~/Scripts/App/ViewModels/chooseproductviewmodel.js" type="text/javascript"></script>  
<script src="~/Scripts/App/ViewModels/placeorderviewmodel.js" type="text/javascript"></script>  
<script src="~/Scripts/App/ViewModels/deliveryviewmodel.js" type="text/javascript"></script>  
<script src="~/Scripts/App/ViewModels/confirmationviewmodel.js" type="text/javascript"></script>  
<script src="~/Scripts/application.js" type="text/javascript"></script>

# Navigation

* Hashbangs, Hashes: Normalmente los hashes los utilizamos dentro de un anchor link: ejm miarticulo.html#capitulo3 para que luego de hacer click y cargue la nueva página, no se muestre el inicio de la página sino desde donde comienza el contenido del capitulo3. Podemos hacerlo mismo para navegar en una misma página sin recargarla.

Ahora que tenemos una aplicación que funciona bien cargando el contenido utilizando AJAX podemos aprovechar estas urls con hash para brindar una navegación a nuestra aplicación: a nivel de cliente podemos detectar cada vez que se haya producido un cambio en el hash de la url, entonces si un usuario escribe en el navegador una url con hash, podemos obtener cual de ese hash y cargar el contenido por AJAX directamente.

¿Qué logramos con todo esto? Logramos que nuestro contenido que es cargado con AJAX pueda ser identificado a través de una URL. A las urls que sirven para este propósito se les llama normalmente AJAX URLS.

Qué son los hashbangs? Como ya es posible contar con urls para contenido dinámico, google permite que ese contenido pueda ser indexado por su buscador, pero con unas cuantas condiciones: la primera de ellas que si queremos que un contenido por ajax sea indexado tenemos que indicarlo colocando un signo de admiración o bang al lado del hash, el resto de condiciones ya son más complejas que requieren tener mucho cuidado con la implementación y salen de nuestro tema principal.

Un clásico ejemplo sobre la utilización de hasbangs es Twitter.

Si bien nosotros podríamos implementar manualmente todo este tema de los hash, existen varias librerías que nos permiten hacer esto de manera más sencilla.

* Routing: Seguramente los que han trabajado con Rails, sympony, play framework me entienden. Cuando hablamos de routing nos referimos en este caso a una tabla en la cual definimos varias urls y para cada urls un handler, de tal manera que apenas se ingrese o modifique la url, se buscará en esta tabla algún campo que coincida con la nueva url y se ejecutará su handler.
* HTML 5 History, nos ofrece un API para manipular la historia de navegación desde el cliente, parte de esa api ya esta disponible desde versiones anteriores y lo que se ha hecho es agregar más funcionalidad para poder utilizar AJAX con las URLS sin necesidad de los hash. History.js nos permite utilizar HTML5 History más fácilmente y si el navegador no soporta HTML5 genera automáticamente urls con hash.

## Codigo

**DeliveryViewModel**

var DeliveryViewModel = function () {  
  
   
    Sammy(function () {  
        this.get('#delivery', function () {  
            self.showChooseProduct();  
        });  
   
        this.get('#delivery/confirmation', function () {  
            self.showConfirmation();  
        });  
   
        this.get('#delivery/:productId', function () {  
            self.showPlaceOrder(this.params.productId);  
        });  
   
        this.get('', function () { this.app.runRoute('get', '#delivery'); });  
    }).run();  
}

**ChooseProductViewModel**

var ChooseProductViewModel = function () {  
   
    self.goToPlaceOrder = function (product) {  
        location.hash = 'delivery/' + product.id;  
    };  
  
};

**PlaceOrderViewModel**

var PlaceOrderViewModel = function (productId) {  
  
    self.postOrder = function () {  
        OrdersDataSource.create(self.order, function () {  
            location.hash = 'delivery/confirmation';  
        });  
    };  
};

**ConfirmationViewModel**

var ConfirmationViewModel = function () {  
    var self = this;  
    self.goToProducts = function () {  
        location.hash = 'delivery';  
    };  
};

**Referencias**

<script src="~/Scripts/Lib/sammy.min.js" type="text/javascript"></script>

# Offline Data

# Offline Resources

Como ya hemos visto HTML5 introduce nuevos métodos para habilitar a las aplicaciones a funcionar sin conexión a la red. Otra de estas formas es el “Application Cache”.

El Application Cache es un conjunto de archivos (HTML, JS, CSS) que son almacenados en la cache del navegador y son utilizados cuando no haya conectividad a la red.

La lista de archivos que deben y no debe ser cacheados, es controlada a través de un archivo denominado “manifest”, es un archivo con formato plano en el cuál debemos seguir ciertas convenciones para agregar esta información.

Este archivo está compuesto de 3 secciones:

* CACHE

Declaramos explícitamente las URLs de los recursos que deben ser cacheados.

* FALLBACK

Dos columnas, la primera un patrón de URL que represente 1 o más recursos online y la segunda que recurso devolver si se intenta acceder a ese recurso de manera offline.

* NETWORK

Que recursos no deben ser cacheados y solo deben estar disponibles online.