# Getting Started

For more information on this topic, see: <http://docs.kendoui.com/getting-started/what-is-kendo>

Getting and adding KendoUI to your project.

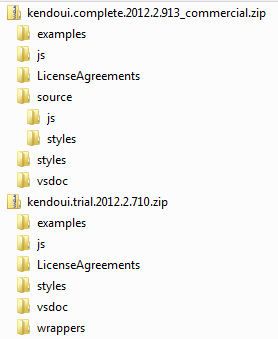
## Getting KendoUI

For more information on this topic, see: <http://docs.kendoui.com/getting-started/downloading-kendo>

KendoUI is divided into three parts: Web, Mobile, and DataViz.

The preferred method of getting KendoUI is to browse to <http://www.kendoui.com/download> and download KendoUI as a .zip file.

Depending on whether you are using the Trial version, or have purchased the full Commercial version, you will have a .zip file that follows one of these directory structures:



The \js and \styles folders in the root of the .zip file contain the “minified” versions of the KendoUI .js and .css files. If you have the commercial version of KendoUI, you may also notice the \source folder. This folder also contains \js and \styles folders, but these are the full, non-minified versions of the .js and .css. These would typically be used for local development and debugging, but the minified versions are the ones that should be deployed in a real application. For that reason, we will use the minified files in the remainder of this example project.

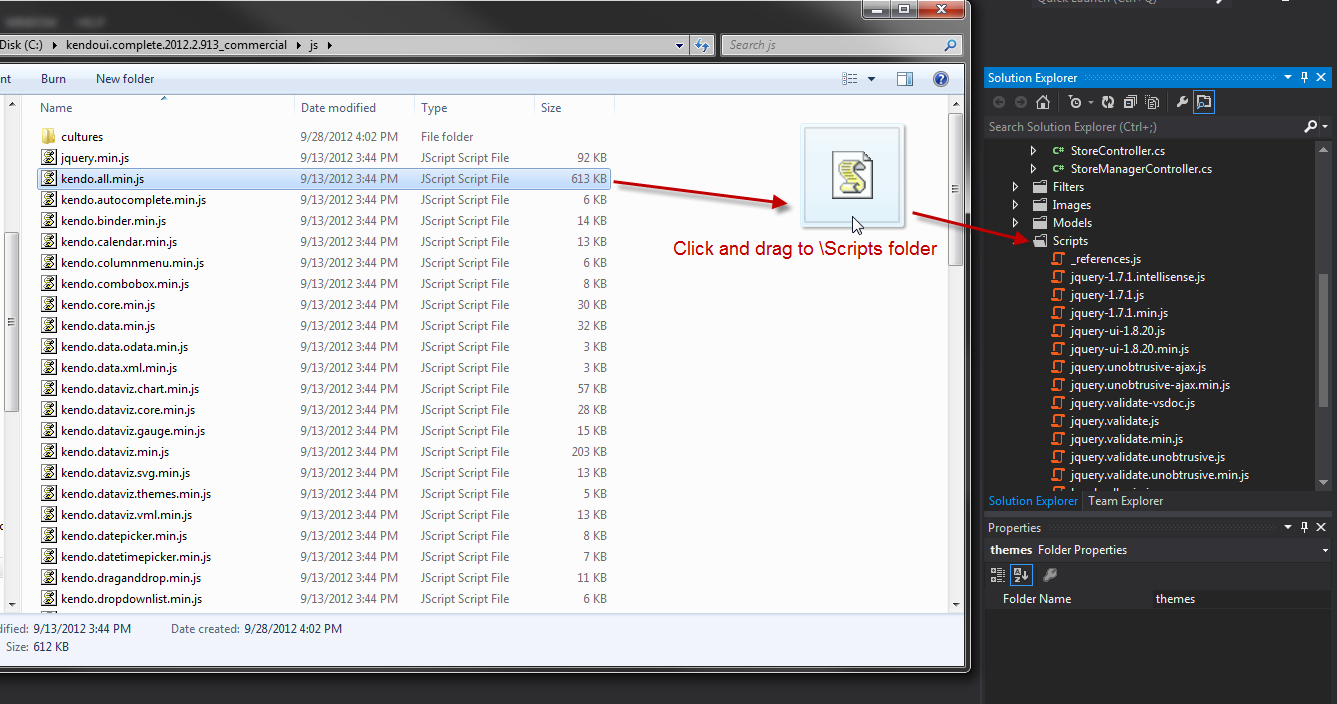
## Adding KendoUI to your ASP.NET MVC Project

For more information on this topic, see: <http://docs.kendoui.com/getting-started/using-kendo-with/aspnet-mvc/introduction>

To add KendoUI to your project, you just need to copy the files you need into your project. For a standard ASP.NET MVC project, files from the Kendo .zip \js folder are copied to ASP.NET MVC’s \Scripts folder, and files from the Kendo .zip \styles folder are copied to ASP.NET MVC’s \Content\themes\base folder.

If you know exactly which Kendo controls and styles you are going to use in your project, you can include individual JavaScript files for just those controls. This can provide a much smaller download for your end users. Optionally, you can use the single file “kendo.all.min.js” which contains the minified files for all of Web, Mobile, and DataViz. You do not need to add kendo.all.min.js *and* each individual kendo.\*.min.js file. Include just one or the other. For more information, see: <http://docs.kendoui.com/getting-started/javascript-dependencies>

When copying the files to your ASP.NET MVC project, you can simply drag and drop the files you want from Windows Explorer to the Solution Explorer in VisualStudio. This will automatically copy the files to your project’s directory, and add them to the project.



## VisualStudio Integration and Templates for ASP.NET MVC

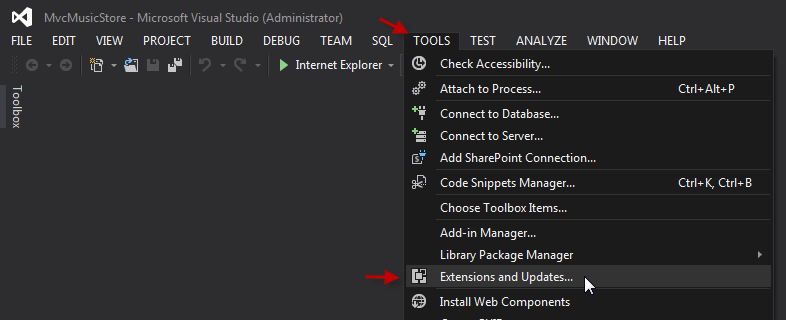
For more information on this topic, see: <http://docs.kendoui.com/getting-started/using-kendo-with/aspnet-mvc/introduction>

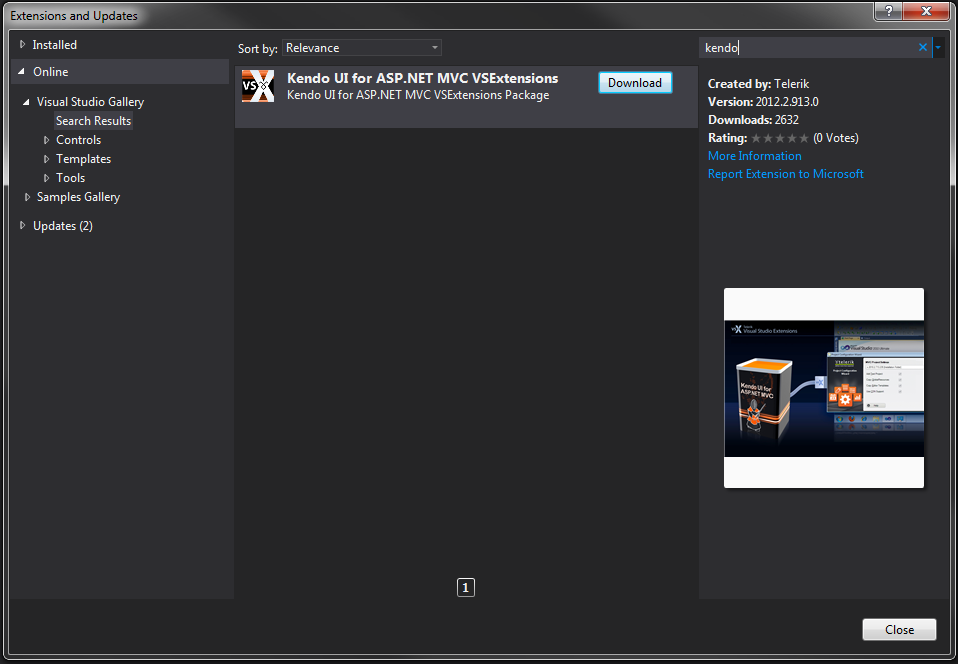
Optionally, Kendo UI provides Kendo UI Complete for ASP.NET MVC.

Kendo UI Complete for ASP.NET MVC is a set of HTML helpers which help you configure Kendo UI widgets by using server-side code in ASP.NET MVC applications. It also provides ready-to-use editor templates based on various Kendo UI widgets.

There is also a VisualStudio Extension available in the VisualStudio Gallery here: <http://visualstudiogallery.msdn.microsoft.com/65b78c2c-951e-43a8-bae7-f9039f59fb9b>

Or it can be added through VisualStudio by selecting “Tools | Extensions and Updates…” from the main menu, then searching for “kendo”.





If you would like to know more about this set of utilities, please refer to the above link.

## jQuery

Kendo is dependent on jQuery. When Kendo JavaScript files are included in an HTML page, jQuery must be included before Kendo.

# Creating a Consumer Web Site with KendoUI Web

## Include Kendo files in rendered Views

Add the CSS and JS to all rendered pages by adding entries to \Views\Shared\\_Layout.cshtml. We need to make sure that Kendo’s JavaScript files come after jQuery. For styling, we include kendo.common.min.css, plus one of the Kendo UI Themes. In this example we are using kendo.default.min.css, which is the Default theme.

<head>

<meta charset="utf-8" />

<title>@ViewBag.Title - Kendo UI Music Store</title>

<link href="~/favicon.ico" rel="shortcut icon" type="image/x-icon" />

<meta name="viewport" content="width=device-width" />

@Styles.Render("~/Content/css")

<link href="~/Content/kendo.common.min.css" rel="stylesheet" /> // <-- Added Here.

<link href="~/Content/kendo.default.min.css" rel="stylesheet" /> // <-- Added Here.

@Scripts.Render("~/bundles/modernizr")

</head>

<body>

...

@Scripts.Render("~/bundles/jquery")

<script src="~/Scripts/kendo.all.min.js"></script> // <-- Added Here.

@RenderSection("scripts", required: false)

</body>

It would also be good practice to use the MVC4 “bundler” capabilities to merge multiple Kendo JS files into a single file for client download. Since we are using the single kendo.all.min.js file in this example, we do not need to perform the bundling.

## Creating the main layout Header and Footer

The layout for our Music Store is contained in \Views\Shared\\_Layout.cshtml

The desired layout for our header is:



The two elements to note here are the menu and search box. For these elements, we will be using the Kendo [Menu](http://demos.kendoui.com/web/menu/index.html) and [AutoComplete](http://demos.kendoui.com/web/autocomplete/index.html) controls.

The basic markup for this header is:

<header>

<section>

@Html.Partial("\_LoginPartial")

</section>

<section>

<nav>

<ul id="menu">

</ul>

</nav>

<button />

<input type="text" />

<section>

</header>

### Add the main menu

Let’s use the Kendo Menu control to add a navigation menu to the Music Store.

To start, we can define the menu items as a <ul> with HTML and Razor:

<ul id="menu">

<li>@Html.ActionLink("Home", "Index", "Home")</li>

<li>Genres

@{Html.RenderAction("GenreMenu", "Store");}

</li>

<li>About</li>

<li>@Html.ActionLink("Contact", "Contact", "Home")</li>

</ul>

Converting this to a Kendo menu is easy. We just add the JavaScript:

$(document).ready(function () {

$("#menu").kendoMenu();

});

This uses a jQuery selector to get our <ul> tag with the id “menu,” and turns it into a Kendo UI menu. This type of kendo usage is sometimes called “explicit” widget initialization, since the element is directly targeted in JavaScript (We will see examples of implicit initialization, data-\* attributes and the MVVM pattern later on, but for our first control, let’s keep it simple.)

Next we can move the Genres list into a pull-down menu. Any <li> elements that in turn contain another list <ul> element are treated as a pull-down item. Let’s start with an empty pull-down menu item for our Genres:

<li>Genres

<ul>

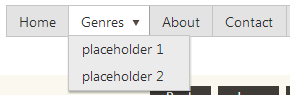
<li>placeholder 1</li>

<li>placeholder 2</li>

</ul>

</li>

This creates the pull-down item:



We can replace our “placeholder” <li> elements with an MVC partial that loads the list of Genres:

<li>Genres

<ul>

@{Html.RenderAction("GenreMenu", "Store");}

</ul>

</li>

This partial outputs a list of <li> elements that work as links to each Genre:

@model IEnumerable<MvcMusicStore.Models.Genre>

<ul>

@foreach (var genre in Model)

{

<li>@Html.ActionLink(genre.Name,

"Browse", "Store",

new { Genre = genre.Name }, null)

</li>

}

</ul>

Another ability of the Kendo Menu control is adding any content into a drop-down menu item. It does not have to be a list of selectable items, like the Genres. To add content to a menu item, Include a <ul> element, with a single <li> element. Within that element, use a <div> to specify the content. For example, let’s add this content for out “About” menu item:

<li>About

<ul>

<li>

<div class="k-content menu-content">

<img src="~/Images/kendo-logo.png" />

<p>Try Kendo UI yourself in the interactive Kendo UI Dojo!</p>

<p><a href="http://www.kendoui.com/dojo.aspx" target="\_blank">Launch Kendo UI Dojo</a></p>

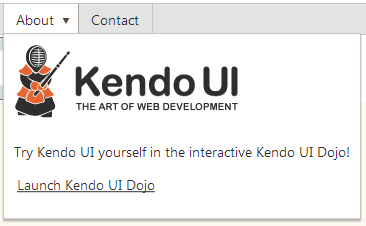
</div>

</li>

</ul>

</li>

This creates the <div> content displayed in a drop-down menu item:



### Adding the Search AutoComplete box.

Our site header contains a text box that can be used to search for artists and albums. We are going to use a Kendo AutoComplete control to display a list of possible matches to the user as they type.

This will also bind to a [remote data source](http://demos.kendoui.com/web/datasource/remote-data.html) to retrieve the auto-complete items.

We start with a normal <input> element, and initialize it with Kendo:

<input id="main-search"/>

$("#main-search").kendoAutoComplete({

filter: 'contains',

minLength: 3,

dataTextField: "Title",

placeholder: "Search music...",

dataSource: {

type: "odata",

serverFiltering: true,

serverPaging: true,

pageSize: 20,

transport: {

read: {

url: "/Api/Albums",

dataType: "json"

},

parameterMap: function (options, type) {

var paramMap = kendo.data.transports.odata.parameterMap(options);

delete paramMap.$inlinecount;

delete paramMap.$format;

return paramMap;

}

},

schema: {

data: function (data) {

return data;

},

total: function (data) {

return data.length;

}

}

},

template: "<img src='${data.AlbumArtUrl}' /><span>${data.Title}</span><span>${data.Artist.Name}</span>"

});

This JavaScript configures the input box to bind to a remote data source. Some of the notable parameters are:

*filter: ‘contains’* – Specifies that autocomplete results just need to contain the entered text, anywhere. The default is ‘starts with’.

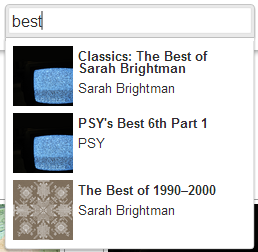
*minLength: 3* – Indicates that the autocomplete dropdown will not display until the user has entered at least 3 characters.

*dataTextField: ‘Title’* – The name of the field in the data that is searched to find autocomplete results. In this example, we are showing autocomplete results for albums whose titles contain the text entered by the user.

*placeholder: “Search music…” –* This is the text thatis displayed in the text box as a placeholder, until the user clicks into the input box.

*dataSource: {}* – Here we configure the source for our autocomplete data. We have specified the URL to our WebAPI for Albums as the source. There is a lot going on in this data source, but most of it is enabling server-side filtering using OData.

*Template: “…”* – Each autocomplete item in the dropdown will be a <li> element, and within that element, include this template. If not specified, Kendo will put the text of the field specified by the *dataTextField* property into the <li>. We can use this template to make much nicer looking results. In this case, we are including the album cover art.



## Creating the Home page.

For the main content of the “Home” page, we want to display a main banner image rotator, and 2 groups of Albums. One will be our “top sellers,” and the other is a “featured artist.” For this, we will be using the Kendo [ListView](http://demos.kendoui.com/web/listview/index.html) widget. For this view, we will also use a “declarative” approach to initializing the widgets, using the data-\* attributes, and the [Kendo MVVM](http://demos.kendoui.com/web/mvvm/index.html) framework.

### Displaying items in a ListView (with MVVM, templates, and binding to a remote data source)

To implement our lists of “featured artist” and “top selling” albums, we can start with some simple HTML markup to represent the 2 ListView widgets:

<section>

<h3>Featured Artist: <span data-bind="text: featuredArtistName"></span></h3>

<div data-role="listview" data-bind="source: featuredArtistAlbums" data-template="album-template">

</div>

</section>

<section>

<h3>Top Selling Albums:</h3>

<div data-role="listview" data-bind="source: topSellingAlbums" data-template="album-template">

</div>

</section>

We have set the “data-role” of each <div> to “listview” so that Kendo knows to transform these <div>s into ListView widgets. Each album will look the same between the 2 lists, so we chose to share a template. Both <div>s get the attribute “data-template” set to “album-template”. In addition, we also need to add the template itself:

<script id="album-template" type="text/x-kendo-template">

<div class="album">

<img src="#: AlbumArtUrl #" />

<span>#: Title #</span>

<span>#: Artist.Name #</span>

<span>#: Price #</span>

</div>

</script>

Note the special “type” of “text/x-kendo-template”. This is required for Kendo to be able to resolve the template. Also the “id” matches the template indicated by the <div> tags. More information on templates can be found [here](http://demos.kendoui.com/web/templates/index.html).

Finally, the “data-bind” property contains our list of binders that will be applied to this widget. In this case we are using the “source” binder, and binding it to the “featuredArtistAlbums” property of the view model. This means we need to create our view model in JavaScript, which is:

$(document).ready(function () {

var featuredArtist = "Metallica";

var viewModel = kendo.observable({

featuredArtistName: featuredArtist,

featuredArtistAlbums: new kendo.data.DataSource({

type: "odata",

serverFiltering: true,

serverPaging: true,

pageSize: 5,

filter: {

field: "Artist/Name",

operator: "eq",

value: featuredArtist

},

transport: {

read: {

url: "/Api/Albums",

dataType: "json"

},

parameterMap: function (options, type) {

var paramMap = kendo.data.transports.odata.parameterMap(options);

delete paramMap.$inlinecount;

delete paramMap.$format;

return paramMap;

}

},

schema: {

data: function (data) {

return data;

},

total: function (data) {

return data.length;

}

}

}),

topSellingAlbums: new kendo.data.DataSource({

transport: {

read: "/Api/Albums?popular=5"

}

})

});

kendo.bind("#body", viewModel);

});

Our viewModel is a Kendo [Observable](http://docs.kendoui.com/api/framework/observableobject) object, which facilitates the updating of properties and notifying the view when it needs to be redrawn. The last line of JavaScript tells kendo to apply bindings between the viewModel and the HTML element with the ID “body”, using normal jQuery selector syntax. In our view model, we have set “Metallica” as our featured artist. This text string will be shown in our <h3> header element, due to the data-binding:

<h3>Featured Artist: <span data-bind="text: featuredArtistName"></span></h3>

The featuredArtistAlbums and topSellingAlbums properties of the view model are both set to Kendo [DataSources](http://demos.kendoui.com/web/datasource/remote-data.html) that will pull remote data from our MVC4 WebAPI controllers. In the case of the featuredArtistAlbums data source, there is additional code specified that enabled server-side filtering by artist name.

### Image Rotator (with a custom binding)

The final item we need to add to your main page is an image rotator. Kendo does not actually provide an image rotator widget, however, the Kendo MVVM bindings are extendable, and so we will write our own [custom binding](http://demos.kendoui.com/web/mvvm/custom.html) to handle this.

We can start by setting up a property on our view model to hold the URLs for the images we want to rotate:

var viewModel = kendo.observable({

bannerImages: [

"/Content/Images/banner1.jpg",

"/Content/Images/banner2.jpg"

]

The bannerImages property is just a simple array of image URLs. Next we make our custom binding:

kendo.data.binders.rotateImages = kendo.data.Binder.extend({

init: function (element, bindings, options) {

kendo.data.Binder.fn.init.call(this, element, bindings, options);

var binding = this.bindings.rotateImages;

var target = $(element);

binding.rotateDelay = target.data("rotate-delay");

binding.imageIndex = 0;

binding.doImageRotation = function () {

var imageArray = binding.get();

var nextImageUrl = imageArray[binding.imageIndex];

target.fadeTo('slow', 0, function () { target.attr('src', nextImageUrl).fadeTo('slow', 1); });

binding.imageIndex++;

if (binding.imageIndex >= imageArray.length) {

binding.imageIndex = 0;

}

};

},

refresh: function () {

var binding = this.bindings.rotateImages;

binding.imageIndex = 0;

binding.doImageRotation();

binding.interval = setInterval(binding.doImageRotation, binding.rotateDelay);

},

destroy: function () {

var binding = this.bindings.rotateImages;

clearInterval(binding.interval);

}

});

The binding looks complicated, but functionally, this is fairly simple. All it does is use a JavaScript setInterval() to periodically change the “src” attribute of an <img> element. However, it does demonstrate a way to retain data in the binding by adding properties to the object instance, and a way to pass in additional data, in this case the delay time. We use a second data-\* property on the element, “data-rotate-delay”, and the custom binding can read this data property with jQuery $(element).data(“rotate-delay”).

Now we can make our HTML that uses our custom binding:

<img data-bind="rotateImages: bannerImages" data-rotate-delay="5000" />

We have indicated that this <img> element will be bound to our custom “rotateImages” binding, using the “bannerImages” property of the view model as its data. Through use of the additional data-\* attribute, we have also indicated that our image will update every 5000 milliseconds (5 seconds).

## Creating the Genre list page

### Display albums in a genre (with a Paging ListView)

When we navigate into a specific genre, we want to display all the albums that apply. We will do this with a [ListView](http://demos.kendoui.com/web/listview/index.html) widget, the same as on the Home page, but this time we will add pagination.

Just like with the Home page, we set up a simple <div> element to contain the list of albums, with its data-role set to “listview”, its source set to the DataSource property of our MVVM view model, and a template that will be used for each album.

<script id="album-template" type="text/x-kendo-template">

<div class="album" data-bind="click: viewAlbumDetails">

<img src="#: AlbumArtUrl #" />

<span class="title" data-bind="text: Title"></span>

<span class="artist" data-bind="text: Artist.Name"></span>

<span class="price" data-bind="textFormatted: Price" data-format="c"></span>

</div>

</script>

<section>

<div data-role="listview" data-bind="source: albums" data-template="album-template"></div>

</section>

Then we set up our view model in JavaScript. We will again use OData to filter our albums server-side, as we had on the Home page for our “Featured Artist” list. This time, we will filter on a Genre ID instead of Artist Name.

var viewModel = kendo.observable({

albums: new kendo.data.DataSource({

type: "odata",

pageSize: 20,

serverFiltering: true,

serverPaging: false,

filter: {

field: "GenreId",

operator: "eq",

value: genreId

},

transport: {

read: {

url: "/Api/Albums",

dataType: "json"

},

parameterMap: function (options, type) {

var paramMap = kendo.data.transports.odata.parameterMap(options);

delete paramMap.$inlinecount;

delete paramMap.$format;

return paramMap;

}

},

schema: {

data: function (data) {

return data;

},

total: function (data) {

return data.length;

}

}

})

});

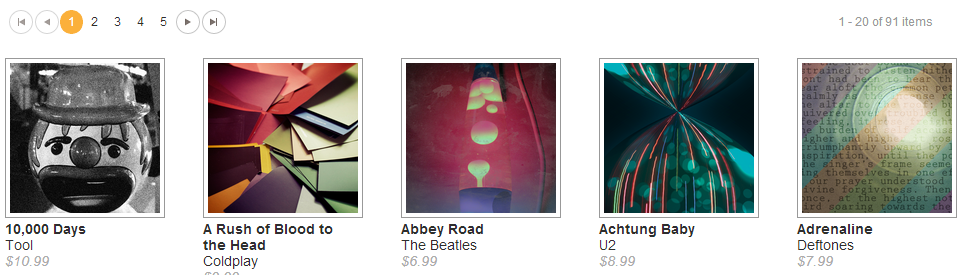
Note that we have set the “pageSize” to 20, so we will display 20 results per page. In this example, we are going to perform pagination on the client by setting the serverPaging property to “false”. This means that the client will pull all albums in the genre from the server, and then split it into pages of 20. This is because in this example we are using MVC4 WebAPI’s OData extensions, which do not support the full OData implementation. One of the items that it does not support is the OData $inlinecount parameter, which Kendo needs to do pagination correctly. Loading all the albums for the genre is a workaround for this example, where we know that none of the data sets are very large. However in a full production application, we recommend either adding $inlinecount support to MVC4 WebAPI, using a data source with a full OData implementation, or implementing your own paging server-side.

This will only display the first 20 albums, but will not show any paging controls. The pagination buttons are actually a separate widget in Kendo. We can add them with another <div> places next to the ListView:

<div data-role="pager" data-bind="source: albums"></div>

<div class="clearfix" data-role="listview" data-bind="source: albums" data-template="album-template"></div>

The data-role is set to “pager”, and the source is bound to the same DataSource property as the ListView widget. Since the Pager and ListView widget are bound to the same DataSource, they stay perfectly in sync. As the Pager widget updates the page on the DataSource, the ListView is notified of the change, and redraws itself for the selected page.



### Displaying the Genre Name (Binding to plain HTML with AJAX loaded data)

Most of our displayed elements up to this point have been actual Kendo widgets. However, the Kendo MVVM framework also has the ability to bind to plain HTML elements. To demonstrate this, we will display the Genre name at the top of the page.

Start by adding an HTML element for the genre name. In this example we will use an <h3> tag.

<h3 data-bind="text: genre.Name"></h3>

We are still using the data-bind attribute to specify our bindings, but we do not have any data-role specified. This means that the element will just use the bindings, but not be turned into a widget. On our view model, this element is bound to “genre.Name”, so we need to add that to the JavaScript. However, up until now we had used a Kendo DataSource to retrieve our remote data. With Kendo, only widgets can bind to a DataSource. Non-widgets need to use plain JavaScript objects that are not a DataSource. You cold bind to an array or another JavaScript object, but in this case we need to get our Genre data from WebAPI remotely. To accomplish this, we will use a regulat jQuery AJAX call to load our genre as JSON, and add this returned object to our viewModel.

var viewModel = kendo.observable({

albums: new kendo.data.DataSource({

/\* code omitted... \*/

}),

genre: null, // <-- this will hold our Genre object, once loaded.

});

// Load the Genre data from the server.

$.ajax({

url: "/Api/Genres/" + genreId,

type: "GET",

dataType: "json",

success: function (data) {

viewModel.set("genre", data);

}

});

kendo.bind("#body", viewModel);

The view model was given a “genre” property that was initially set to null. The actual data was loaded using jQuery asynchronously. Since the data load is asynchronous, but MVVM will notify the view of any update to the “genre” property, we do not need to do any extra work to handle the asynchronous load. In other words, the <h3> may initially bind to the “null” value and display nothing, but when the load finishes and .set() is called, the genre property will be updated, and the text binding on the <h3> will be notified to update the text.

## Displaying Album details (with multiple Window widgets)

Whenever we need to show the details for an album, either from a selection in the “search” auto-complete box, clicking an album on the Main page from the “featured” lists, or clicking an album from the Genre list page, the album details will be opened in a kendo [Window](http://demos.kendoui.com/web/window/index.html) widget.

Normally, a placeholder <div> element would be included in the page, and then the window created with the JavaScript:

$('#windowPlaceHolderElement').kendoWindow({...});

However, for this example, we are going to open a new window for each album clicked, instead of reusing a single element. This means the user can have multiple album detail windows open at once. To do this, a <div> element will be added to the page dynamically each time an album is clicked, and removed when the window is closed.

In the Main and Genre pages, we used a template to render each Album. To respond to click events, we can add a “click” binding, and give it a function on the viewModel:

<script id="album-template" type="text/x-kendo-template">

<div class="album" data-bind="click: viewAlbumDetails">

And add the function to the view model in JavaScript:

var viewModel = kendo.observable({

/\* existing code omitted...\*/

// this will handle the album clicks.

viewAlbumDetails: function (e) {

}

$.ajax({

url: "/Api/Albums/" + e.data.AlbumId,

type: "GET",

dataType: "json",

success: function (data) {

that.\_openWindow("album-details-template", that.\_getAlbumDetailsViewModel(data));

}

});

});

The function will then need to add a new <div> element to the page to be the window, transform the element into a Window widget, and specify a custom close handler that will remove the <div> from the page when the window is closed. This functionality was placed in a helper function to help separate the window opening and closing functionality from the rest of the view model. Also, each album details window will have its own view model.

this.\_openWindow = function (template, viewModel) {

// Create a placeholder element.

var window = $(document.createElement('div'));

// Apply template to the placeholder element, and bind the viewmodel.

var templateHtml = $(document.getElementById(template)).html()

window.html(kendo.template(templateHtml)(viewModel));

kendo.bind(window, viewModel);

// Add window placeholder to the body.

$('body').append(window);

// Turn placeholder into a Window widget.

window.kendoWindow({

width: "400px",

title: viewModel.data.Title,

resizable: false,

close: function () {

// When the window is closed, remove the element from the document.

window.parents(".k-window").remove();

}

});

// Center and show the Window.

window.data("kendoWindow").center();

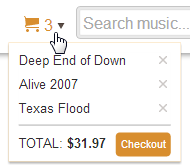
window.data("kendoWindow").open();

};

This JavaScript starts by creating a new <div> element to be our new window widget. We then make some function calls into the Kendo framework to translate a template. In this case, instead of passing in a string of HTML to use as the template, I wanted to pass in the name of a template, much the same as the MVVM “data-template” attribute. Then the window <div> element is appended to the document body. Next the new element is transformed into a Kendo Window widget by calling .kendoWindow() on it. We also provide a function to the “close” event of the window that will remove this <div> from the document when the window is closed. Finally, we call .center() and .open() to position and display the window.

## Creating the Shopping Cart Menu (with a custom widget)

One of the interesting aspects of the Music Store application, from a design perspective, was the desire to display the contents of the shopping cart, along with the total purchase price, in a drop-down panel under the shopping cart button.



This was a good way to demonstrate a custom widget. In this case, a “composite widget” will be created, which is simply a widget composed of other widgets. Here, a normal Kendo [Menu](http://demos.kendoui.com/web/menu/index.html) widget will be used as the cart button and to display the drop-open panel, and a [ListView](http://demos.kendoui.com/web/listview/index.html) widget will be placed within the menu to display each of the albums currently in the cart. Also, the widget will be bound to a kendo [DataSource](http://demos.kendoui.com/web/datasource/index.html), which contains the cart data.

I will let the specifics of the custom widget be an exercise for the reader, instead of copy/pasting every line of code here, but the basic structure of the widget is:

(function ($) {

var CartMenu = Widget.extend({

// options that are available to the user when initializing the widget

options: {

name: "CartMenu",

},

// method called when a new widget is created

init: function (element, options) {

var that = this;

Widget.fn.init.call(that, element, options);

},

refresh: function () {

},

});

ui.plugin(CartMenu);

})(jQuery);

In its simplest form, a widget basically has 2 major functions defined; “init()” is called whenever a widget is created, for example when calling $(“#someElement”).kendoCartmenu() or when the MVVM framework encounters a data-role=”cartMenu”. “refresh()” is then called any time the widget needs to be updated. Typically, you set up and initialize everything for your widget in the init() function, including manipulating the DOM to insert any extra HTML elements that may be required. The refresh() function then updates the HTML elements as needed.

Each widget should be placed in its own JavaScript file, to clearly separate it from the rest of the code. In this example, the widget is in kendo-cart-menu-widget.js.

*For more information on creating custom Kendo Widgets, Burke Holland has excellent posts on the KendoUI blog,* [*here*](http://www.kendoui.com/blogs/teamblog/posts/12-04-03/creating_custom_kendo_ui_plugins.aspx)*, and* [*here*](http://www.kendoui.com/blogs/teamblog/posts/12-04-10/creating_a_datasource_aware_kendo_ui_widget.aspx)*.*

# Creating an Administrative Panel with KendoUI Kendo Grid and DataViz Charts

# Creating a Consumer Mobile Site KendoUI Mobile