

Migration Guide

ActionScript 3.0 to JavaScript and Sencha Frameworks

Table of Contents

Introduction	1
Setting Up Sencha Frameworks for Development	1
Instantiating Objects	4
Event Listeners	5
Using xtype	7
Classes and Packages	9
Extending Classes	11
Getters and Setters	12
Firing Events	14
Statics	16
Member Access Control	18
Event Scope in Classes	19
Custom xtype	21
Dynamic Loading and Deployment	22
Deployment with Sencha SDK Tools	24
Conclusion	26

Introduction

As the web evolves, many ActionScript developers find themselves investigating HTML5 and JavaScript as possible next steps forward. With its Ext JS and Sencha Touch frameworks, Sencha provides a smooth migration path for application developers experienced with ActionScript's robust class system. Ext JS creates JavaScript-based applications for desktop browsers, and Touch creates apps for mobile devices. They both take JavaScript's flexible prototype-based language features and sculpt them into a more familiar and structured class system that offers a foundation from which to continue effectively building rich web applications.

In this paper, we'll learn how Sencha frameworks assist with object-oriented programming in JavaScript, from creating and manipulating objects, to defining and extending classes. A variety of examples, both in ActionScript and JavaScript, demonstrate how easily we can translate typical language constructs from one environment to the other.

Finally, we'll learn the difference between a Sencha framework application's development and production environments. We'll discover how Sencha frameworks can easily switch between loading intuitively-organized class files dynamically and loading the more compact, packaged version of our code that is ideal for an optimized production scenario.

Setting Up Sencha Frameworks for Development

Before we start learning Sencha's approach to object-oriented programming techniques, let's set up a simple development environment from which we can try out the upcoming examples.

What You'll Need

- Sencha Touch or Ext JS.
- A web server running locally on your computer.
- A modern web browser. Chrome and Safari are recommended.

Let's start by downloading one of the Sencha frameworks. The upcoming code examples will work with both Sencha Touch and Ext JS. Follow the instructions below for the Sencha framework that you choose.

Instructions for Sencha Touch

- Download the Sencha Touch SDK.
- Place the contents of the unzipped SDK into your web server's document root. We recommend using a simple one-click installer like WAMP or MAMP.
- Open your web browser and point it to <http://localhost/touch> (or wherever your web server is configured to serve from) and you should see the Sencha Touch Welcome page.
- Next to the touch folder, in your web server's document root, create two new files. One named `index.html` and the other named `app.js`.
- In the file named `index.html`, add the following markup:



```
1 <!doctype html>
2 <html>
3 <head>
4     <meta charset="UTF-8">
5     <title>AS3 to JS/Sencha Migration</title>
6     <link rel="stylesheet" type="text/css" href="touch/resources/css/sencha-touch.css">
7     <script type="text/javascript" src="touch/sencha-touch-debug.js"></script>
8     <script type="text/javascript" src="app.js"></script>
9 </head>
10 <body>
11 </body>
12 </html>
```

Instructions for Ext JS

- Download the Ext JS SDK.
- Place the contents of the unzipped SDK into your web server's document root. We recommend using a simple one-click installer like WAMP or MAMP.
- Open your web browser and point it to <http://localhost/extjs> (or wherever your web server is configured to serve from) and you should see the Ext JS Welcome page.
- Next to the extjs folder, in your web server's document root, create two new files. One named `index.html` and the other named `app.js`.

```
1 <!doctype html>
2 <html>
3 <head>
4     <meta charset="UTF-8">
5     <title>AS3 to JS/Sencha Migration</title>
6     <link rel="stylesheet" type="text/css" href="extjs/resources/css/ext-all.css">
7     <script type="text/javascript" src="extjs/ext-debug.js"></script>
8     <script type="text/javascript" src="app.js"></script>
9 </head>
10 <body>
11 </body>
12 </html>
```

Continued Instructions for All Frameworks

In the file named app.js, add the following JavaScript code:

```
1 Ext.onReady(function()
2 {
3     console.log( "Ready to go!" );
4 });
```

This is the most basic code needed to get an application based on either Ext JS or Sencha Touch up and running. The function passed to `Ext.onReady()` will be called once the core Sencha framework is loaded. The JavaScript code from the upcoming examples should be placed in this function (or in a separate class file, when applicable).

Side Note: Like `trace()` in ActionScript, `console.log()` in a browser is an easy way to display simple text for debugging purposes.

To ensure that everything is working correctly, open `http://localhost/` in your browser and open the JavaScript console to view our log message. In Google Chrome, click the Wrench icon to the right of the URL input box, go to the Tools submenu, and choose “JavaScript Console”. In Safari, open the Develop menu and select “Show Error Console”. If the Develop menu is not visible, open Safari’s preferences, go to the Advanced section, and check “Show Develop menu in menu bar”. For other browsers, please check the official documentation for details.

Instantiating Object

In an ActionScript or Flex application, we use the new keyword to create an object.

```
1 var button:Button = new Button();
```

We create objects in Sencha by calling Ext.create().

```
1 Ext.create( "Ext.Button" );
```

Notice that the class name is referenced as a string. This allows us to take advantage of a Sencha framework's ability to load classes dynamically during development. We can organize JavaScript files into an intuitive package structure, and Sencha loads the files as needed. We'll learn more about this later, when we start creating our own classes.

If you'd like, you can save a reference to the button as a variable:

```
1 var button = Ext.create( "Ext.Button" );
```

Next, let's customize the button. We want to add a label and display the button in our application. We'll start with the ActionScript version again:

```
1 var button:Button = new Button();  
2 button.label = "Click Me";  
3 addChild( button );
```

In the Sencha version, we do the same by passing a configuration object to `Ext.create()`. A configuration object allows us to customize things like properties, styles, and event listeners.

```
1 var button = Ext.create("Ext.Button", {
2     text: "Click Me",
3     renderTo: Ext.getBody()
4 });
```

In this case, we use the `renderTo` configuration to place the button into the body element of the document. In later examples, we'll learn additional ways to specify where to display Buttons and other UI components.

Event Listeners

Let's continue by adding an event listener to the button so that our code can react to a user pressing it. With `ActionScript`, we call the `addEventListener()` function and pass in a listener function to handle the event.

```
1 function onClick( event:MouseEvent ):void {
2     button.label = "Clicked";
3 }
4
5 var button:Button = new Button();
6 button.label = "Click Me";
7 button.addEventListener( MouseEvent.CLICK, onClick );
8 addChild( button );
```

Sencha provides several approaches for adding event listeners. The approach that's most similar to the one used in `ActionScript` is this:

```
1 var button = Ext.create("Ext.Button", {
2     text: "Click Me",
3     renderTo: Ext.getBody()
4 });
5 button.addListener( "click", function( e ) {
6     this.setText( "Clicked" );
7 });
```

The “click” event is defined by `Ext.Button` in `ExtJS`. With `Sencha Touch`, you should use the “tap” event instead.

In `JavaScript`, it’s common practice to define anonymous functions inline. If we want, we can also follow the approach used in the previous `ActionScript` example. We can name the function, and then we can reference it by name in the `addListener()` call.

We can also add event listeners in the configuration for a new object. Actually, this is usually more common than calling `addListener()`. Many classes offer a handler configuration as a shortcut to add a listener for an object’s default event (usually, the event that is most commonly used). In the case of the `Button` class, it’s the `click` event:

```
1 var button = Ext.create("Ext.Button", {
2     text: "Click Me",
3     renderTo: Ext.getBody(),
4     handler: function( e ) {
5         this.setText( "Clicked" );
6     }
7 });
```

Alternatively, the listeners configuration allows us to add the `click` event listener, along with any of the other events dispatched by our button:

```
1 Ext.create("Ext.Button", {
2     text: "Click Me",
3     renderTo: Ext.getBody(),
4     listeners: {
5         click: function( e ) {
6             this.setText( "Clicked" );
7         }
8     }
9 });
```

Simply reference one or more events by name using keys in the object passed to the listeners configuration. The value for each key is the associated listener function. As before, change the event name to `tap` if you’re using `Sencha Touch` instead of `ExtJS`

In the three previous Sencha examples, notice that we use the `this` keyword with the call to `setText()`. In these cases, `this` refers to the button. The default scope for an event handler is the dispatcher's scope. In later examples, we'll learn how the `this` keyword can be bound to a different scope.

Important: In ActionScript, it's usually possible to omit the `this` keyword when referencing a member in the current scope. However, in JavaScript, you must always use the `this` keyword because the function's scope is bound at runtime instead of compile time.

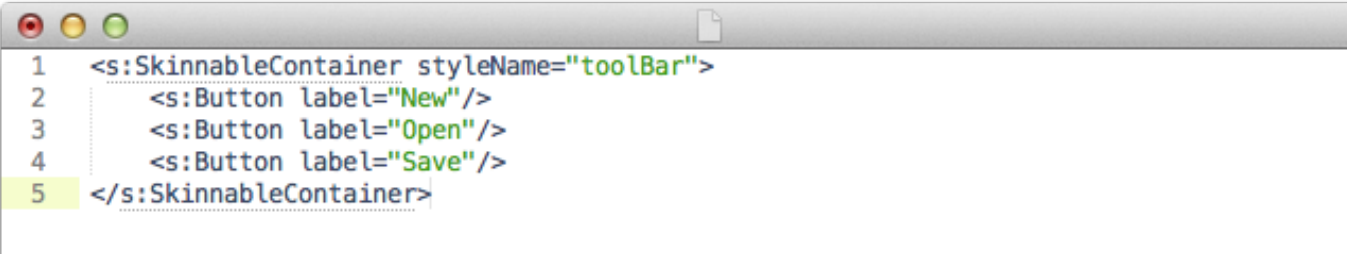
Using xtype

When we call `Ext.create()` to instantiate a new component, the framework creates the component immediately. However, many rich applications have a deeply nested structure where parts of the UI aren't visible on startup. For instance, an application may have a floating window that opens only when the user presses a button to edit the application's settings. Similarly, many applications organize content using tabbed pages, and the content displayed by background tabs may not be immediately visible. In these cases, we can improve startup performance by deferring instantiation of the UI until it is needed.

Sencha's approach for deferring a component's instantiation is to replace a call to `Ext.create()` with the object's configuration. To the configuration, we add an `xtype`, which is a shorthand name for the class that the framework will eventually instantiate when the object is finally needed. We will use `xtypes` most often when defining children of containers (and even containers within containers).

In many ways, the use of `xtypes` is a more declarative approach for UI, very similar to Flex's MXML. Components are declared using simple JavaScript object literals, which is just a tiny step away from JSON, a common data format used often interchangeably with XML for similar tasks. Likewise, MXML defers component instantiation in a very similar way, as many Flex developers know from experience with tweaking creation policies.

To demonstrate the simplest usage of `xtypes`, let's create a toolbar that contains several buttons. In Flex, we might create a toolbar in MXML like this:



```
1 <s:SkinnableContainer styleName="toolBar">
2     <s:Button label="New"/>
3     <s:Button label="Open"/>
4     <s:Button label="Save"/>
5 </s:SkinnableContainer>
```

Note that the Flex framework doesn't provide a container specifically meant to be used as a toolbar. However, one can easily be created by combining a `SkinnableContainer` with an appropriate skin. In this case, let's assume that a CSS file in the project defines an appropriate skin for `SkinnableContainer.toolbar`.

Sencha has a first-class toolbar container, so we'll use that in our example:

```
1 Ext.create( "Ext.Toolbar", {
2     renderTo: Ext.getBody(),
3     items: [
4         {
5             xtype: "button",
6             text: "New"
7         },
8         {
9             xtype: "button",
10            text: "Open"
11        },
12        {
13            xtype: "button",
14            text: "Save"
15        }
16    ]
17 });
```

Notice that we define our root container, the toolbar, with `Ext.create()`, but the buttons inside the toolbar are defined using only their configuration objects with an extra `xtype` configuration added in. If we were to place another container into the toolbar, we could nest `xtype` declarations indefinitely. In addition to deferred instantiation, we've made our code a little bit shorter than if we made several `Ext.create()` calls, while keeping it just as readable.

Because we know that all of the toolbar's children will be buttons, we can actually simplify this code a bit more:


```
1 Ext.create( "Ext.Toolbar", {
2   renderTo: Ext.getBody(),
3   defaultType: "button",
4   items: [
5     {
6       text: "New"
7     },
8     {
9       text: "Open"
10    },
11    {
12      text: "Save"
13    }
14  ]
15 });
```

The `defaultType` configuration on the toolbar lets us omit the `xtypes` for the buttons. Any configuration without an `xtype` will inherit the default, so we can still take advantage of `defaultType` when most, but not necessarily all, of the toolbar's children are buttons. We can even omit the `defaultType`, in this case, because a toolbar's default value for `defaultType` is already "button".

When we describe how to define new classes with Sencha in the next section, we'll also look at how custom `xtypes` can be added to components.

Classes and Packages

Like `ActionScript`, Sencha's type system includes full support for organizing classes within packages. This includes complete organization of packages as directories in the file system. Sencha's dynamic class loading automatically builds URLs for each class based on the package structure.

In `ActionScript`, the fully-qualified class name `com.example.utils.Counter` maps to the file `com/example/utils/Counter.as`. Likewise, in a Sencha application, the fully-qualified class name `Example.util.Counter` maps to the file `Example/util/Counter.js`.

Defining Classes

The following examples show how to create a counter, that is a class that counts. This simple class will allow us to learn many of the capabilities Sencha offers for defining classes. Let's start by reviewing `ActionScript`'s `class` keyword:

```
1 package com.example.utils {
2     public class Counter {
3         public function Counter() {
4             }
5
6         public var count:int = 0;
7
8         public function addOne():void {
9             count++;
10        }
11    }
12 }
```

In Sencha frameworks, we call `Ext.define()` to define a class. It's very similar to how we use `Ext.create()` to instantiate an object. The first argument is the fully-qualified class name, and the second argument is a configuration object that defines properties and methods, including overrides.

```
1 Ext.define( "Example.util.Counter", {
2     constructor: function() {
3         this.callParent( arguments );
4         return this;
5     },
6     count: 0,
7     addOne: function() {
8         this.count++;
9     }
10 });
```

ActionScript uses the class name as the name of the constructor function, but in Sencha frameworks, the constructor function is defined with the generic name `constructor` in all classes. You may return `this` at the end of the constructor, but it's not required. As with ActionScript, you may omit the constructor entirely if you're not running any code in it.

To call any overridden super class function (like constructor, in this case), use `this.callParent()`. Pass the function's arguments object, or an Array if you want to override the arguments passed to the super class. Sencha will automatically determine which function to call on the super class.

We don't specify a class to extend here, so Sencha frameworks automatically extend the `Ext.Base` class, which includes the core functionality for all classes. This is much like how ActionScript always extends `Object` if no super class is specified.

To instantiate our new class, we use `Ext.create()`, as we did before to create a button.

```
1 var counter = Ext.create( "Example.util.Counter" );
2 counter.addOne();
3 console.log( counter.count ); // 1
4 counter.addOne();
5 console.log( counter.count ); // 2
```

Extending Classes

To make our simple counter a little more useful, it should dispatch an event when the count value changes. Let's start by extending the appropriate class to make event dispatching possible.

```
1 package com.example.utils {
2     import flash.events.EventDispatcher;
3     public class Counter extends EventDispatcher {
4         public function Counter() {
5             }
6
7         public var count:int = 0;
8
9         public function addOne():void {
10             count++;
11         }
12     }
13 }
```

In ActionScript, we extend the EventDispatcher class. Now, the Sencha version:

```
1 Ext.define( "Example.util.Counter", {
2     extend: "Ext.util.Observable",
3     constructor: function() {
4         this.callParent( arguments );
5         return this;
6     },
7     count: 0,
8     addOne: function() {
9         this.count++;
10    }
11 });
```

Add the extend configuration to specify the super class. In Sencha, the Observable class allows us to fire events.

Getters and Setters

Ideally, we want to dispatch an event any time that the count value changes. We could dispatch the event in the addOne() function, but we want the count value to be public so that it can be changed arbitrarily from outside the Counter class. What we need here is a setter function that not only sets count, but also dispatches an event.

```
1 package com.example.utils {
2     import flash.events.EventDispatcher;
3     public class Counter extends EventDispatcher {
4         public function Counter() {
5         }
6
7         private var _count:int = 0;
8
9         public function get count():int {
10             return _count;
11         }
12
13         public function set count(value:int):void {
14             _count = value;
15         }
16
17         public function addOne():void {
18             count++;
19         }
20     }
21 }
```

In ActionScript, we use the `get` and `set` keywords to define getter and setter functions. In Sencha frameworks, we define the config value, and then we call `initConfig()` in the constructor to initialize the values through their setters:

```
1 Ext.define( "Example.util.Counter", {
2     extend: "Ext.util.Observable",
3     config: {
4         count: 0
5     },
6     constructor: function( config ) {
7         this.initConfig( config );
8         this.callParent( arguments );
9         return this;
10    },
11    addOne: function() {
12        this.setCount( this.getCount() + 1 );
13    },
14    setCount: function( value ) {
15        this.count = value;
16    }
17 });
```

Sencha frameworks automatically generate getters and setters for all values defined in config, if we don't define them manually. In this case, the function `getCount()` will be generated for our class, and we'll be customizing the `setCount()` setter ourselves.

Let's update our logging code to use the new getter:

```
1 var counter = Ext.create( "Example.util.Counter" );
2 counter.addOne();
3 console.log( counter.getCount() ); // 1
4 counter.addOne();
5 console.log( counter.getCount() ); // 2
```

Firing Events

Now that we've added a getter and setter to count, let's fire a "change" event from the setter.

```
1 package com.example.utils {
2     import flash.events.Event;
3     import flash.events.EventDispatcher;
4
5     [Event(name="change",type="flash.events.Event")]
6
7     public class Counter extends EventDispatcher {
8         public function Counter() {
9             }
10
11         private var _count:int = 0;
12
13         public function get count():int {
14             return _count;
15         }
16
17         public function set count(value:int):void {
18             _count = value;
19             dispatchEvent( new Event( Event.CHANGE ) );
20         }
21
22         public function addOne():void {
23             count++;
24         }
25     }
26 }
```

In ActionScript, we can create a new Event object and then pass it to `dispatchEvent()`. We could define the event name with a "change" string, but the Event class defines a `CHANGE` constant. ActionScript supports special metadata for events, so we've defined the change event with metadata above the class.

There are very similar additions to a Sencha class:

```
1 Ext.define( "Example.util.Counter", {
2     extend: "Ext.util.Observable",
3     config: {
4         count: 0
5     },
6     constructor: function( config ) {
7         this.addEvents({
8             change: true
9         });
10        this.initConfig( config );
11        this.callParent( arguments );
12        return this;
13    },
14    addOne: function() {
15        this.setCount( this.getCount() + 1 );
16    },
17    setCount: function( value ) {
18        this.count = value;
19        this.fireEvent( "change" );
20    }
21 });
```

First, we call `addEvents()` in the constructor to define which events will be fired, similar to how `ActionScript` uses metadata. Adding your events is required, and a runtime error will be thrown if you try to dispatch an event that hasn't been added. In `Sencha` frameworks, to dispatch an event, use the `fireEvent()` function and pass a string value for the event name. You may pass additional parameters to `fireEvent()` after the event name and they will be passed as arguments to the listeners.

Now, instead of manually logging the count after each call to `addOne()`, let's add an event listener to log the count:

```
1 var counter = Ext.create( "Example.util.Counter" );
2 counter.addListener( "change", function() {
3     console.log( counter.getCount() );
4 });
5 counter.addOne();
6 counter.addOne();
```

Statics

Imagine an application that tracks votes in an election, with several Counter instances. Each counter tracks the votes in a specific district, but we're also interested in the total number of votes among all of the districts. To implement this, we might add a static variable to our class that includes the total of all counters.

```
1  package com.example.utils {
2      import flash.events.Event;
3      import flash.events.EventDispatcher;
4
5      [Event(name="change",type="flash.events.Event")]
6
7      public class Counter extends EventDispatcher {
8          public static var totalCount:int = 0;
9
10         public function Counter() {
11         }
12
13         private var _count:int = 0;
14
15         public function get count():int {
16             return _count;
17         }
18
19         public function set count(value:int):void {
20             totalCount += value - _count;
21             _count = value;
22             dispatchEvent( new Event( Event.CHANGE ) );
23         }
24
25         public function addOne():void {
26             count++;
27         }
28     }
29 }
```


In ActionScript, simply use the static keyword to define a static variable. Similarly, use a configuration value called `statics` in Sencha:

```
1  Ext.define( "Example.util.Counter", {
2      extend: "Ext.util.Observable",
3      config: {
4          count: 0
5      },
6      statics: {
7          totalCount: 0
8      },
9      constructor: function( config ) {
10         this.addEvents({
11             change: true
12         });
13         this.initConfig( config );
14         this.callParent( arguments );
15         return this;
16     },
17     addOne: function() {
18         this.setCount( this.getCount() + 1 );
19     },
20     setCount: function( value ) {
21         if( isNaN( this.count ) ) {
22             this.count = 0; // ensure that count is defined
23         }
24         this.self.totalCount += value - this.count;
25         this.count = value;
26         this.fireEvent( "change" );
27     }
28 });
```

As you can see, use `this.self` to access the class, and with it, the static members. We only define a static variable in this example, but static functions may be defined in the `statics` configuration too, just like non-static functions may appear in the main configuration.

Now, let's also log `totalCount` in our event listener:

```
1 var counter = Ext.create( "Example.util.Counter" );
2 counter.addListener( "change", function() {
3     console.log( counter.getCount(), Example.util.Counter.totalCount );
4 });
5 counter.addOne();
6 counter.addOne();
```

As you can see, statics are accessed with a direct reference to our class.

Member Access Control

To create private variables in Sencha classes, we need to define class members a little bit differently. We'll use local variables and closures in the constructor.

```
1 Ext.define( "Example.util.Counter", {
2     extend: "Ext.util.Observable",
3     statics: {
4         totalCount: 0
5     },
6     constructor: function( config ) {
7         this.addEvents({
8             change: true
9         });
10
11         var count = 0;
12
13         this.getCount = function() {
14             return count;
15         }
16
17         this.setCount = function( value ) {
18             this.self.totalCount += value - count;
19             count = value;
20             this.fireEvent( "change" );
21         }
22
23         this.initConfig( config );
24         this.callParent( arguments );
25         return this;
26     },
27     addOne: function() {
28         this.setCount( this.getCount() + 1 );
29     }
30 });
```

Here, we've defined the count value and its getter and setter in the constructor. We've also removed count from the config for the class because we're defining the getter and setter manually to ensure that the variable remains private.

Because count is a local variable, it becomes inaccessible outside of the Counter class. Similarly, a function that appears within the constructor, but is not assigned to this.functionName, is also considered private, and it is only accessible to other functions defined within the constructor.

We've attached getCount() and setCount() to this in order to make them public. They appear within the constructor, so they also have privileged access to private members. On the other hand, addOne() can appear outside of the constructor because it is only using public APIs. If we needed to access the count variable from within addOne(), it too would need to be defined within the constructor.

Event Scope in Classes

Let's return to events for a moment. In particular, we haven't looked closely at how scope works. Take a moment to review the toolbar example from earlier, displayed below and converted to a class.

```
1 Ext.define( "Example.view.FileToolbar", {
2     extend: "Ext.Toolbar",
3     constructor: function( config ) {
4         config.items = [
5             {
6                 text: "New"
7             },
8             {
9                 text: "Open"
10            },
11            {
12                text: "Save"
13            }
14        ];
15        this.callParent(arguments);
16    }
17 });
```

Notice that we're defining the items configuration in the `constructor()` function. We're doing this because we're going to need access to the toolbar's scope in a moment.

To create a `FileToolbar` instance, use the following code in `app.js`:

```
1 Ext.create("Example.view.FileToolbar",
2 {
3     renderTo: Ext.getBody()
4 });
```

Now, let's add a listener for the "New" button's click event.

```
1 Ext.define( "Example.view.FileToolbar", {
2     extend: "Ext.Toolbar",
3     constructor: function( config ) {
4         config.items = [
5             {
6                 text: "New",
7                 listeners: {
8                     click: this.onNewButtonClick
9                 }
10            },
11            {
12                text: "Open"
13            },
14            {
15                text: "Save"
16            }
17        ];
18        this.callParent( arguments );
19    },
20    onNewButtonClick: function( button, event, eOpts ) {
21        console.log( this.xtype ); // button
22    }
23 });
```

Custom xtypes

So far, we've only talked about a couple of xtypes for built-in framework components. However, the xtype system is fully extensible to include custom components. It's as simple as adding an alias configuration to one of our classes.

Let's take the FileToolbar class from our previous examples, and add an xtype:

```
1 Ext.define( "Example.view.FileToolbar", {
2     extend: "Ext.Toolbar",
3     alias: "widget.filetoolbar",
4     constructor: function( config ) {
5         config.items = [
6             {
7                 text: "New",
8                 listeners: {
9                     scope: this,
10                    click: this.onNewButtonClick
11                }
12            },
13            {
14                text: "Open"
15            },
16            {
17                text: "Save"
18            }
19        ];
20        this.callParent( arguments );
21    },
22    onNewButtonClick: function( button, event, eOpts ) {
23        console.log( this.xtype ); // filetoolbar
24    }
25 });
```

Don't forget the required widget. prefix. In the onNewButtonClick function, you can also see how the console will be updated with the new xtype. Now, we can reference the "filetoolbar" xtype when we create a FileToolbar as a child of another container, such as the main application viewport, below:

```
1 Ext.create( "Ext.Viewport",
2 {
3     items: [
4         {
5             xtype: "filetoolbar"
6         }
7     ],
8     renderTo: Ext.getBody()
9 });
```

However, because the “filetoolbar” xtype is defined in Example/view/FileToolbar.js, Sencha frameworks know that the xtype exists only after the class has been loaded. In app.js, we inform the Sencha framework that the class is required, and it is loaded before the function passed to Ext.onReady() is called:

```
1 Ext.require( "Example.view.FileToolbar" );
2 Ext.onReady( function() {
3     Ext.create( "Ext.Viewport",
4     {
5         items: [
6             {
7                 xtype: "filetoolbar"
8             }
9         ],
10        renderTo: Ext.getBody()
11    });
12 });
```

Dynamic Loading and Deployment

Sencha’s frameworks work in two different ways. During development, classes are saved in separate files and loaded as they’re needed. For production, the Sencha SDK Tools combine and minify JavaScript to load everything all at once. Let’s get more familiar with the capabilities of the dynamic loader, and then we’ll learn how to create a production build.

Sencha’s Dynamic Loader

For the most part, dynamic loading of classes happens unobtrusively and automatically in the background. A few options are available to customize the behavior of the loader. Let’s take a look at a few of them.

Ext.require()

If we add a call to `Ext.require()` before our application starts, we can specify classes to load immediately (rather than as they're needed), including all dependencies. Here, we pass a single class name:

```
1 Ext.require( "Ext.Button" );
```

We can also specify a list of classes:

```
1 Ext.require( [ "Ext.Button", "Ext.Toolbar" ] );
```

A few additional arguments are available, including a callback function to be called when the required classes finish loading and a list of classes to exclude. Please see the `Ext.require()` documentation for more details.

Ext.Loader.setPath()

In many of the examples above, the class named `Example.util.Counter` maps to `Example/util/Counter.js`. If we wanted to change the name of the root folder (perhaps we want to change it to `example` with a lower-case `e`, or maybe we want to put our packages in a `src` folder), we could do it like this:

```
1 Ext.Loader.setPath( "Example", "src/example" );
```

Ext.Loader.setConfig()

Finally, we can set a variety of options on `Ext.Loader` with a single call, like this:

```
1 Ext.Loader.setConfig({
2     disableCaching: true,
3     paths: {
4         "Ext", "src/ext",
5         "Example", "src/example"
6     }
7 });
```

The paths key is used to call `setPath()` for several packages. The `disableCaching` key will manipulate URLs to bypass the browser's file cache. For more information about `Ext.Loader` and its capabilities, please see the complete documentation for `Ext.Loader`.

Deployment with Sencha SDK Tools

During development, it's very useful to have classes separated out into different files and well-formatted for easy reading. Sencha frameworks take care of finding the classes in the package file structure and classes are loading at runtime as they're needed. However, when we deploy to the web, it's a good practice to limit HTTP calls to as few as possible and to make our files smaller to save on bandwidth. We want to limit our impact on server performance and on the time and data limits of our visitors.

The steps for deploying JavaScript applications works a lot like how ActionScript applications are packaged into a single SWF containing compiled bytecode. Generally, we want to put all of our JavaScript code into a single file, and then compress, or minify in web development terms, the code so that it takes up as little space as possible. Sencha SDK Tools includes and uses the open source YUI Compressor to remove whitespace and comments and to obfuscate local variable names.

Get Sencha SDK Tools

- Download and install the Sencha SDK Tools.
- Open a command prompt, such as PowerShell on Windows or Terminal on Mac OS X.
- Run the `sencha` command without arguments to verify that it installed correctly.

Please note: If your machine is running the 64-bit version of Windows, you will need to ensure that the 32-bit version of Java is used to run the following commands. You can open the x86 version of PowerShell, or you can run the following command in regular PowerShell before using the Sencha SDK tools to update your path for the current session only:



```
1 $env:Path += ";C:\Program Files (x86)\Java\jre6\bin"
```

If the Java runtime is installed at a different location on your computer, enter that path instead of the one above.

Building for Production

To start a build, first we need to generate a JSB3 file. Run the following command, pointing to the URL of index.html in your development environment:

```
1 sencha create jsb -a http://localhost/index.html -p app.jsb3
```

This command will create app.jsb3, which we will need in the next step:

```
1 sencha build -p app.jsb3 -d .|
```

The second command generates two files, app-all.js and all-classes.js. The file all-classes.js contains all of your application's classes from all packages. The contents are not minified, so this file can be useful for debugging issues that may arise with the built application that may not be present in the development version. The other file, app-all.js, includes the minified classes and the contents of app.js, our application's main JavaScript file. This is the version that will be deployed to production.

Finally, the production version of the application should use a slightly modified HTML file to load the new scripts.

Sencha Touch Production HTML

```
1 <!doctype html>
2 <html>
3 <head>
4 <meta charset="UTF-8">
5     <title>Sencha Production</title>
6     <link rel="stylesheet" type="text/css" href="extjs/resources/css/ext-all.css">
7     <script type="text/javascript" src="touch/sencha-touch.js"></script>
8     <script type="text/javascript" src="app-all.js"></script>
9 </head>
10 <body></body>
11 </html>
```

Ext JS Production HTML

```
1 <!doctype html>
2 <html>
3 <head>
4     <meta charset="UTF-8">
5     <title>Sencha Production</title>
6     <link rel="stylesheet" type="text/css" href="extjs/resources/css/ext-all.css">
7     <script type="text/javascript" src="extjs/ext.js"></script>
8     <script type="text/javascript" src="app-all.js"></script>
9 </head>
10 <body></body>
11 </html>
```

The app.js development code has been replaced with the concatenated and minified app-all.js and the debug version of the Sencha framework has been replaced by the production version.

Conclusion

As we've seen, Sencha frameworks provide a professional foundation for JavaScript application development that complements the ActionScript developer experience. Key characteristics of developing with Ext JS and Sencha Touch -- including object instantiation, working with events, support for classes and packages, adding statics, defining scope, adding custom components, and dynamic loading -- will provide a familiar migration path for ActionScript developers to follow. ActionScript developers can apply many of the same techniques they've grown accustomed to with ActionScript to the web environment by using Sencha frameworks along with JavaScript and quickly start taking advantage of the increasingly robust capabilities offered by native web technologies.