

Durango Hello World

This article introduces the basic concepts of Durango by walking through the creation of a Durango-enabled application and a couple of reusable components. By the end of this walk-through, you will be able to:

- Create a Flex project that references the Durango libraries
- Create a reusable visual component
- Create a reusable service (non-visual) component
- Enable an application to donate and receive Durango components

You must also have Flex Builder installed.

Create a Flex Project

This discussion assumes you are using Flex Builder. While you can achieve the same results using just the Flex SDK, doing so is beyond the scope of this article.

To create a Durango project in Flex Builder:

1. Add a new Flex Project (*File>New>Flex Project*)
2. Name the project: `DurangoHelloWorld`
3. Choose **Desktop application** (runs in Adobe AIR)
4. On the third page of the Create Flex project wizard, click the **Library** path tab.
5. Click the **Add SWC** button.
6. Enter or Browse to the Durango ReuseLib.swc. This library file will be found in the Durango installation folder chosen when installing Durango. The default location of ReuseLib.swc is `C:\Program Files\Adobe\Durango\Libraries\ResureLib.swc`. (You can also add the library SWC to an existing project using the Properties dialog.)
7. Click **Finish**

A new Flex project with an MXML file, `DurangoHelloWorld.mxml`, is created. We will edit this file to enable Durango features later in this article, but first we will create a couple of Durango components.

Create a reusable visual component

A reusable component can be dragged from one Durango application to another. Reusable components must implement the Durango `IComposable` interface, but otherwise can be any MXML or ActionScript class. A visual component in this context is simply one that has a GUI or draws to the stage. This article extends an MXML class to create a Durango-enabled component. You can also create a visual component using an ActionScript class.

First, create the class file in Flex Builder:

1. Add a new MXML component (*File>New>MXML Component*):
2. In the New MXML Component wizard, assign the name `DurangoVisualComponent`
3. Choose `TextArea` from the Based on list. The `DurangoVisualComponent.mxml` file is created with `TextArea` as the root tag.

Then, edit the new component to add support for Durango:

1. Add the attribute `implements="atl.reuse.IComposable"` to root tag:
`<mx:TextArea xmlns:mx=http://www.adobe.com/2006/mxml`

```
implements="atl.reuse.IComposable">
```

2. Implement IComposable interface functions to get and set the reusable property:

```
private var r : Boolean = true;

public function get reusable () : Boolean {
    return r;
}

public function set reusable ( b : Boolean ) : void {
    r = b;
}
```

The IComposable interface serves as a marker to indicate that a component can be dragged out of a Durango donor application. The `reusable` property is used by the Durango UI to determine whether a component can currently be dragged out. Your application can set the property to `true` when a component can be dragged out. If the component is always “reusable” you can simply return `true`.

3. Add a connectable property

We want this visual component to display text supplied by another component. Any public, bindable properties can be connected together. However, for Durango to connect two properties automatically when a component is dropped into an application we must use the `[AutoConnect]` metadata tag. With this tag, we can mark a property as a data input (sink), data output (source), or as either. In addition, any property marked with the metadata tag `[Editable]` will be available in the connection editor. Finally, `[AutoConnect(sink, namesMustMatch)]` means that the property will only connect automatically with properties of the same name. Both sink and source properties may be annotated with the `namesMustMatch` tag value.

The property for which we want to enable connections is the `htmlText` property of the `TextArea`. Since this property is inherited, and we can't add metadata to the super class, we must override the property get and set functions in the Durango component. We can then add the metadata tag to the override functions:

```
[AutoConnect(sink,source)] //can connect as both an input
and an output
public override function set htmlText (val : String):void {
    super.htmlText = val;
}

public override function get htmlText () : String {
    return super.htmlText;
}
```

Note that in the case of the `htmlText` property, the super class already adds the bindable metadata, so our child class functions don't need to be tagged with `[Bindable]`.

Metadata for properties

You can use the following Durango-defined metadata tags on properties:

- **[AutoConnect]** - indicate that a component can connect to other components. If a compatible property is available when a component is dragged into an application, Durango will connect the two properties automatically.
- **[Persistent]** - indicate that the value of a property should be saved when a component is dragged out of the application.
- **[Editable]** - indicate that the property should appear in the Durango properties dialog that can be accessed when a Durango application is in design mode.

The final component MXML should look like the following:

Your finished component MXML should look like the following:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:TextArea xmlns:mx=http://www.adobe.com/2006/mxml
    implements="at1.reuse.IComposable">

    <mx:Script>
        <![CDATA[
            // Control dynamic reusability...
            private var r : Boolean = true;

            public function get reusable () : Boolean {
                return r;
            }

            public function set reusable ( b : Boolean ) : void {
                r = b;
            }

            [AutoConnect(sink,source)]
            public override function set htmlText ( val : String ) : void {
                super.htmlText = val;
            }
            public override function get htmlText () : String {
                return super.htmlText;
            }
        ]]>
    </mx:Script>
</mx:TextArea>
```

Before we can use the component, we must add it to a Durango-enabled application. But first, we will create a non-visual service component to serve as the source of the data to be displayed by our visual component.

Create a service (non-visual) component

Service components are non-visual components and are typically used to provide data to visual components. This example uses a very simple service component that supplies an HTML or text value. This value can be edited on the Durango property pane, but is otherwise static.

Because service components don't draw a user interface, Durango displays them as boxes in a special service tray when an application is in design mode. This allows a user to interact with a service component or drag it to another application. If you add a `label` property to the class, the value is displayed on the box shown for the component.

As with a visual component, a service component must implement the `IComposable` interface. It must also implement the `Flex IMXMLObject` interface, which allows Flex to assign an ID to the component.

First, create the `ActionScript` class in Flex Builder:

1. Add a new ActionScript class (*File>New>ActionScript Class*):
2. In the New ActionScript class wizard, assign the name `DurangoServiceComponent`
3. In the **Interfaces** area, add `Interfaces.IComposable` and `mx.core.IMXMLObject`
4. Click **Finish** to create the `DurangoServiceComponent.as` file.

Next, implement the `IComposable` Interface. You can use the same code used earlier for the visual component:

```
private var r : Boolean = true;
public function get reusable () : Boolean {
    return r;
}

public function set reusable ( b : Boolean ) : void {
    r = b;
}
```

Then, implement the `IMXMLObject` interface. To do this we need to add an `id` property and set its value in the `initialized()` method:

```
[Bindable]
[Inspectable]
public var id : String = null;

public function initialized (doc : Object, id : String) : void {
    this.id = id;
}
```

Finally, add an `editable`, `persistent` `sourceText` property that will automatically connect to our visual component. To do this, we add a `String` variable and mark it with the appropriate metadata tags:

```
[Bindable]
[Editable]
[Persistent]
[AutoConnect(source)]
public var sourceText : String = "Hello World";
```

While we are at it, let's add a `label` property so that our component shows its name when the application is in design mode:

```
[Bindable]
[Editable]
[Persistent]
public var label : String = "Text Source";
```

Before moving to the next step, we must add the `[DesignModeOnly]` metadata tag to our service class. This tag indicates that the class has no visual properties. Without this metadata tag, the property window will generate errors when it tries to access display object properties. Add the tag above the class declaration:

```
[DesignModeOnly]
public class
```

```
DurangoServiceComponent implements IComposable, IMXMLObject
{...}
```

The final component ActionScript class

Your finished ActionScript class for the service component should look like the following:

```
package {
    import atl.reuse.IComposable;
    import mx.core.IMXMLObject;

    [DesignModeOnly]
    public class DurangoServiceComponent implements IComposable, IMXMLObject {
        [Bindable]
        [Inspectable]
        public var id : String = null;

        public function initialized (doc : Object, theId : String) : void {
            id = theId;
        }

        [Bindable]
        [Editable]
        [Persistent]
        public var label:String = "Text Source";

        [Bindable]
        [Editable]
        [Persistent]
        [AutoConnect(source)]
        public var sourceText:String = "Hello World";

        private var _reusable:Boolean = true;
        public function set reusable(b:Boolean):void {
            _reusable = b;
        }

        public function get reusable():Boolean {
            return _reusable;
        }
    }
}
```

Enable an application to donate and receive components

To put our components together, we need a Durango-enabled application to host them.

Types of Durango applications

A Durango-enabled application can be one of the following types:

- **Receiver** application – the application can receive reusable components. Such applications can use components to add or customize functionality, or can serve as a host container with all or most functionality provided by reusable components.
- **Donor** application – the application allows its components to be dragged to a different receiver. A donor application might have its own purpose, and also provide components that can be used outside the original application context, or it might simply serve as a catalog of reusable components.
- **Both** – the application can both receive and donate components.

The Durango Hello World application we are creating is both a donor and a receiver. You can drag the visual and service components into another Durango application. You can also drag other components into the Durango Hello World application.

The basic steps to enable Durango behavior are:

- Add a container to receive or donate reusable components.
- Add a `ReusableServicesCollection` component to hold any service components, if desired. This container is only displayed when the application is in design mode.
- Create and initialize a `ComposableDragSupport` object, identifying a container for visual components and the services component.
- Add the `ToolTrayHost` control. The `ToolTrayHost` control adds a fly-out panel on the left edge of the application that gives the user access to configuration commands.

Add a container for visual components

You can use any Flex container to hold the visual components. Durango Hello World uses `Canvas` from the standard Flex Libraries. To use `Canvas`, add the package to the namespace declarations of the `WindowedApplication` tag and add the `GradientCanvas` tag to the file, assigning an id, as well as color and size values. Place the `DurangoVisualComponent` inside this container, as well:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx=http://www.adobe.com/2006/mxml
    layout="absolute"
    xmlns:reuse="atl.reuse.*" xmlns:local="*">
  <mx:Canvas id="reuseArea" width="100%" height="100%">
    <local:DurangoVisualComponent id="display"
      width="100%" height="50%"
      editable="false"/>
  </mx:Canvas>
</mx:WindowedApplication>
```

Add a ReusableServicesCollection component

For service components to be visible in the application design mode, the components must be placed in a `ReusableServicesCollection` component. The `ReusableServicesCollection` component is located in the Durango `atl.reuse` package. Add the `ReusableServicesCollection` tag to the file, assigning an id. Place the `DurangoServiceComponent` inside this collection:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
    layout="absolute"
    xmlns:reuse="atl.reuse.*"
    xmlns:local="*">
  <reuse:ReusableServicesCollection id="services">
    <local:DurangoServiceComponent id="staticText"/>
  </reuse:ReusableServicesCollection>

  <mx:Canvas id="donorArea" width="100%" height="100%">
    <local:DurangoVisualComponent id="display"
      width="100%" height="50%"
```

```

        editable="false"/>
    </mx:Canvas>
</mx:WindowedApplication>

```

Add the ToolTrayHost control

The ToolTrayHost supplies the fly-out panel containing the controls that allow a user to toggle the application design mode. To add the ToolTrayHost control to the application insert the following tag:

```
<reuse:ToolTrayHost/>
```

Create the ComposableDragSupport object

To enable Durango support, you must create and initialize a Durango ComposableDragSupport object. The ComposableDragSupport class is located in the `atl.reuse` package. Add an `<mx:Script>` tag to the application containing the following code:

```

<mx:Script>
    <![CDATA[
        import atl.reuse.ComposableDragSupport;
        public var dragSupport : ComposableDragSupport = null;
        private function initApp () : void {
            dragSupport = ComposableDragSupport.createDragSupport(donorArea,
                                                                    this.nativeWindow, services);
        }
    ]]>
</mx:Script>

```

The `createDragSupport()` method is a factory function that creates the ComposableDragSupport objects. The Durango Hello World application passes the following parameters to the method:

- `tl:DisplayListContainer` - the container for the visual Durango components.
- `w:NativeWindow` - the NativeWindow
- `services:ReusableServicesCollection` - the component containing the non-visual Durango components

The other parameters of the `createDragSupport()` method are left with their default values.

Note, for donor-only applications that cannot receive dragged components, you can also use the `createDragSupportDonor()` method, which doesn't permit the application to be reconfigured at runtime.

Connect the pieces together

If we run the application at this point, we would get a blank text area. All that remains is to link the service and visual components so that data can flow between them. To do this we use a data binding expression in the assignment of a property on the visual component. Add the attribute assignment `htmlText="{staticText.sourceText}"` to the `<DurangoVisualComponent/>` tag. Then add a call to

Tool tray options

Durango Hello World does not set any tool tray options, but you can add additional controls by setting the following attributes to true:

- **offerConfigTools** – adds tools for saving and loading the application configuration.
- **offerCFPB** – adds an option for creating a Flex Builder project from the current application configuration. (offerConfigTools must also be true.)
- **dragSupportInstance** – tells the ToolTrayHost where to find the instance of ComposableDragSupport in the application.

`dragSupport.setupComplete()` to the `initApp()` method to give Durango a chance to find the bound properties.

The final application class

Our final `WindowedApplication` class now looks like this:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute"
  initialize="initApp()" xmlns:local="*" xmlns:reuse="atl.reuse.*">
  <mx:Script>
    <![CDATA[
      import atl.reuse.ComposableDragSupport;

      public var dragSupport : ComposableDragSupport = null;

      private function initApp () : void {
        dragSupport = ComposableDragSupport.createDragSupport( donorArea,
          this.nativeWindow, services);
        dragSupport.setupComplete();
      }
    ]]>
  </mx:Script>

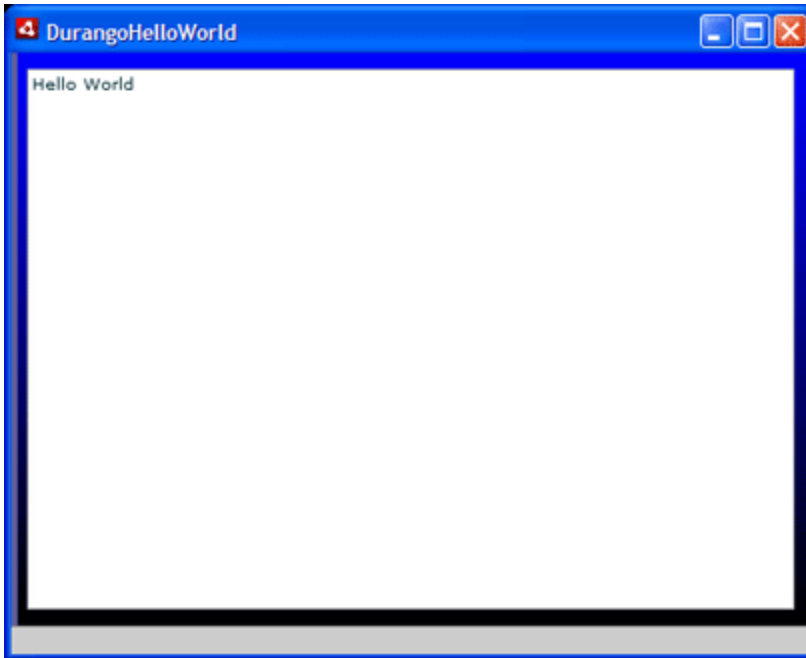
  <reuse:ReusableServicesCollection id="services">
    <local:DurangoServiceComponent id="staticText"/>
  </reuse:ReusableServicesCollection>

  <mx:Canvas id="donorArea" width="100%" height="100%">
    <local:DurangoVisualComponent id="display" htmlText="{staticText.sourceText}"
      top="10" bottom="10" right="10" left="10"
      editable="false"/>
  </mx:Canvas>

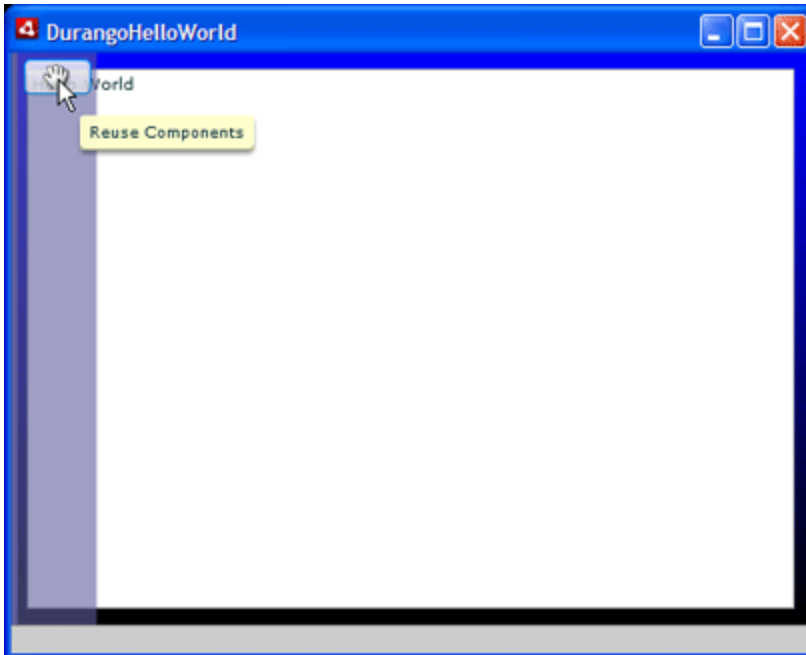
  <reuse:ToolTrayHost/>
</mx:WindowedApplication>
```

Run the application

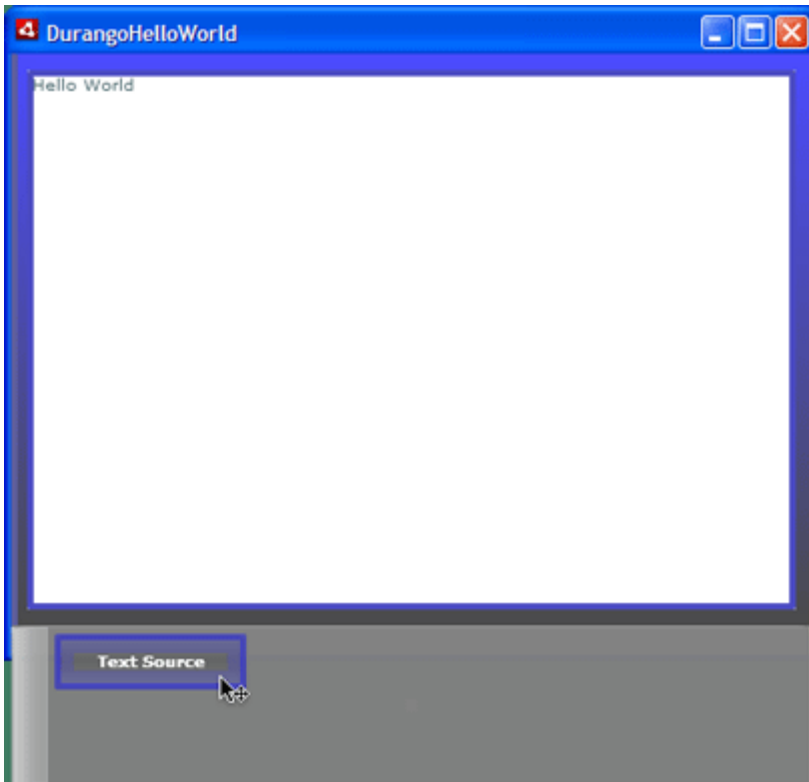
Run the application from Flex Builder. You should see a window that looks like this:



Enter design mode using the tool tray (move the mouse to the left edge of the application):



In design mode, the service components are displayed in the reusable services panel:



To change the text, right-click or Command-click the service component, which is labeled, "Text Source."
The Durango property dialog opens and you can edit the `sourceText` property:

