Flex in a Week, Flex 4.5
Video 4.02: Creating a custom item renderer

In the last video, you learned how to display both string and visual component data in a DataGroup container using the default item renderers.

While these classes are very useful for simple displays, they do not allow you the full range of expressiveness that you might want to employ for your application.

In this video, you will learn the basic building blocks for creating your own custom item renderers.

You will first build an item renderer inline and then as a separate class file.

When you create a custom item renderer, you have full control over the display of the data.

You can use any components and containers you need to in order to manipulate the visual display.

You can create an custom item renderer in two ways:

This first code sample shows the itemRenderer property nested within the DataGroup container.

This second code example shows the itemRenderer property referencing an external class file.

In this video, you will learn how to implement a custom item renderer in both ways.

This is the data for my sample application that I will use in my DataGroup container’s item renderer.

In the last video, I used String and BitmapImage tags for the data.

In this video, I have consolidated the data into generic Object instances with name/value pairs.

You will often see this done to ensure that related data is handled as a unit.

Below the layout property in the DataGroup container, I am adding an itemRenderer tag set.

Within this code, I am creating an instance of a component using the Component class.

This instance cannot be empty and is defining a new scope for the content it contains.

Lastly, I will create an actual ItemRenderer instance using the ItemRenderer class.

Take care to note that the outer item renderer block is named with a lowercase i, which denotes that it is the property of the DataGroup container.

The inner ItemRenderer block, is an instance of the ItemRenderer class, which is why it beings with an initial uppercase I.

The ItemRenderer instance can contain many tags, including these listed here.

You will learn about states in a later video of this Day.

For now, just know that states define different visual aspects of your renderer.

For instance, in a DataGroup container item, you might have a normal state that users see by default, and then a hovered state that users see when they mouse over the item.

To create states simple create a states property block in your ItemRenderer tag block and create one State instance for each state.

You must always define a least the normal state.

Back in Flash Builder, I am creating a states property inside the ItemRenderer instance.

I am creating one state named normal.

This code shows a Button instance inside of the ItemRenderer instance.

You can implement single components like this.

If you want to use multiple components in your renderer, you must place them inside a container.

Remember that the Component tag block represents a new scope.

That means that everything inside of it, is blind to the content outside of it.

To access that data within the item renderer, you must do so through a property named data.

In this code, you can see that a Label control inside of the item renderer, would reference the lastName and firstName values from the dataProvider object by first referencing the data object.

I want to display the employee’s image below his or her name, so I am adding a VGroup container below the states block to lay out this display.

I am creating a Label control that binds the text property to the data.firstName value and a literal space and then the data.lastName value.

The BitmapImage control's source property will bind to the data object's imageFile value, which references the images stored in the images directory of the project.

When I save the file and run the application, you can see that each employee is displayed with the image below the name.

Notice the default hover stated.

Remember that each employee's display represents one rendered iteration over the items in the dataProvider data set.

I want to create a visual distinction between each item by applying a gray background and border, so I am changing the VGroup container into a BorderContainer instance.

I still want the content to be laid out vertically so I'm adding a layout property with a VerticalLayout instance.

I'm also going to horizontally align the content to the center and vertically align it to the middle.

To add the border around each item, I am adding a borderWeight property set to a value of 2 pixels, and a backgroundColor set to a value of #CCCCCC, which is gray.

I am also adding the height and width properties and setting their values to 100%.

When I save the file and run the application, you can see that each item in the DataGroup container is rendered visually distinct from the others.

You can imagine that writing complex item renderer displays inline to the DataGroup container may get unwieldy.

Therefore, you can also create the item renderer display in a separate class file and then simply referencing it from the itemRenderer property.

The class file will extend the ItemRenderer class and all of the other rules you just learned about creating item renderers do apply.

By convention for this training series, I am storing my component files in a components directory.

In the Package Explorer, I am right-clicking on the src folder for this project and selecting New > ItemRenderer.

In the New Item Renderer dialog box, I am typing components for the Package and EmployeeItemRenderer for the Name.

I'm clicking Finish.

In the new EmployeeItemRenderer.mxml file that opened in the Editor view, I'm deleting the generated Label control.

I am returning to the main application file.

Within the DataGroup container instance, remember that the Component tag block creates a new component scope and is essentially like creating a new component class file.

The ItemRenderer tag block is the actual instance of the renderer and is represented by the fact that we are extending this class in the new class file.

I am cutting all the code inside the ItemRenderer block and then returning to the new class file and pasting it.

I am saving the file to maintain my changes and then returning to the main application file.

I am deleting the nested itemRenderer property from the DataGroup container and replacing it with an inline itemRenderer property.

For the value, I am pointing to the components.EmployeeItemRenderer class file.

Remember that the class file is in the components directory.

When I save the file and run the application, you can see that the display is exactly the same.

The only difference is structural.

I have separated the item renderer code from the main application code for easier maintainability.

For your next step, work through the exercise titled “Displaying dynamic data in a custom item renderer”.