Flex in a Week video, Flex 4.5
Video 2.08: Introducing the MVC pattern

Flex data modeling is based on the Model-View-Controller (MVC) design pattern.

In this video, you will learn how to apply this pattern to a simple application.

xIn the MVC pattern, the xmodel refers to the data objects in your application, the xview refers to the user interface elements, and the xcontroller handles the logic for how the model and view interact.

xIn Flex applications, xthe model represents data, like the ArrayCollection objects that contain the data returned from the HTTPService call.

xThe view in an application might be a DataGrid control or other UI component.

xThe controller in a Flex application can be a xspecialized implementation as in many frameworks, like Cairngorm, xbut is often just the main application file in a simple implementation, like the one we will apply in this video.

I will use the Employee Portal: Vehicle Request Form for this example.

The code for this application retrieves data, manipulates it in functions and then displays it in UI components.

Currently all the code used in this example is in one file.

This may suitable for small applications but it is not typical to the majority of applications.

In this video, you will learn how to separate the UI code into a separate custom component, which is the view.

The retrieved data is the model and will be handled in class properties.

The controller, which handles the business logic for the model and view, is the main application file.

I will begin by creating the view.

This means transferring the UI controls to a separate component.

I am creating a new directory in the src folder by selecting New > Folder, which you cannot see because it is outside the recording area, and naming it components.

I am right-clicking on the components directory and selecting New > MXML Component.

In the dialog, I am naming the component VehicleRequestForm.

I am removing the width and height so that the component will size itself to its contents and then I am clicking Finish.

The new custom component file opens in the Editor view, where I am double-clicking on the tab to maximize it.

Back in the main application, I am cutting the UI controls and pasting them into the VehicleRequestForm component.

When I save the component, and the main application file, you can see that the main application file has some errors, as you can see on the right side.

In the Problems view, which I am selecting in the lower right corner, you can see that these errors are related the pickupDate and returnDate.

I am double-clicking on one of the errors to locate the related code.

The DateChooser controls are no longer in the main application so the functions that are referring to them are throwing errors.

I am going to fix those errors by moving the initApp() and dateChangeHandler() functions from the main application into the VehicleRequestForm.

I am cutting these from the main application and then returning to the VehicleRequestForm custom component to create a Script block and paste them in the code.

The initApp() function is responsible for sending the HTTPService call and initializing the event listeners on the DateChooser controls.

The dateChangeHandler() function handles the event when the DateChooser is triggered.

I am changing the name of the initApp() function to just init().

This makes it more appropriate to its purpose in this component.

I am also calling the init() function from the creationComplete event of the custom component.

I am cutting the employeeService.send() method from the init() function and then replacing the initApp() call in the creationComplete event of the main application with this request.

I am saving both files.

These warning messages indicate that the custom component is missing the CalendarLayoutChangeEvent class import statement.

The UI elements do use the CalendarLayoutChangeEvent and the Alert classes so I am copying those import statements from the main application and pasting them to the custom component.

I can actually delete the CalendarLayouChangeEvent from the main application file, but I must leave the Alert class, since it is also used by the data service fault handler.

I am saving both files.

To recap, I have moved the UI controls into the custom component, which is my view.

I have also moved the two functions that manage the events for the DateChooser controls to the component as well.

However, I have made sure that the data request is still managed by the main application by moving the employeeService.send() method back to the creationComplete event of the Application container.

Note that, within the custom component UI elements, the DropDownList control's dataProvider property is bound to the property employees.

The employees property is the ArrayCollection instance created in the main application file that is populated with the returned data from the service call.

I will need to pass this data from the main application to the custom component via a custom component property.

In the Script block after the import statements I am creating a public variable named employees that is datatyped to the ArrayCollection class.

Note that Flash Builder has automatically imported the class for me since I used code assist.

Remember that an ActionScript property that is bound to a UI control must use the [Bindable] metadata directive.

At this point the VehicleRequestForm view is complete.

The component can accept data and populate the DropDownList control.

It also handles the selection of dates from the DateChooser controls.

I am saving the file again.

Now I must integrate the VehicleRequestForm view back into the main application.

In the UI components area of the code, I am adding an instance of the VehicleRequestForm component.

Note that Flash Builder's code assist tool can find the component in the components directory.

When I select it, you can see that Flash Builder places the component in a namespace called components.

When I scroll up to the opening Application tag, you can see that Flash Builder has created this namespace for me.

I am saving the file and running the application.

The DropDownList control does not have any data in it.

This is because I've created the employees property in the custom component, but I haven't used it in my code to send data to the component.

Back in the main application file, I am adding the employees property to the custom component and then adding curly braces for the binding.

Now what do I bind to?

Remember that this employees property is a property of the VehicleRequestForm custom component.

Here it is in the component code.

However, the main application file also has a property named employees.

It is instantiated towards the top of the Script block and then populated with the return data in the result handler.

This is the property that I am binding to.

Even though the employees property of the VehicleRequestForm and the employees property of the main application are named the same, they are *not* the same variables.

The main application employees property is scoped to only exist in the main application.

The VehicleRequestForm component's employees property can only be referenced from that custom component's scope.

I am referencing the application's employees property in the binding.

I am saving the file and running the application again.

When I select the DropDownList control, I can now see that it is populated with data.

I have successfully implemented a simple MVC design pattern, where my view is the VehicleRequestForm custom component, my model is the data stored in the employees property, and my controller is the business logic in the main application file.

For your next step, work through the exercise titled “Separating the model, view and controller”.