Apache Royale
Starting from a blank file

30.09.2020 20:15 CEST

Carlos Rovira & Andrew Wetmore
Objective

Create a TODOMVC application based on http://todomvc.com/
Features

- Jewel Component Set
- MVC
- Crux (microframework)
  - Beans & BeanProvider
  - IoC (Inversion of Control)
  - DI (Dependency Injection)
  - Event Handling
- Theme customization

- Strand & Beads
- View States
- List Item Renderers
- Data Binding
- Routing
- LocalStorage with AMF encoding (automatic encoding/decoding)
Journey

Let’s go!
Creating and building the starter Apache Royale Jewel Crux Project.

Laying out the interface
Creating the initial application visuals.

Setting up the MVC
MVC Design Pattern is the main organization in the example.

Adding a Todo Item
How to wire the use case of “adding an item to the list”.

Drawing each list item
Let’s see how renderers work inside a list.

Styling, Storage & Routing
Final touches to complete the example.
Let’s go!

Creating and building the starter Apache Royale Jewel Crux Project.
Create a project with a Maven Archetype

Create a folder and use the following command to create a Royale **Jewel** Application project with **Crux** support from scratch:

```
mvn archetype:generate
-DarchetypeGroupId=org.apache.royale.framework
-DarchetypeArtifactId=royale-jewel-application-archetype
-DarchetypeVersion=0.9.8-SNAPSHOT

```

Example configuration:

Define value for property 'groupId': com.carlosrovira.examples
Define value for property 'artifactId': todomvc
Define value for property 'version' 1.0-SNAPSHOT: :
Define value for property 'package' com.carlosrovira.examples.todomvc: :

```
- src
  - main
    - resources
    - royale
      - com
        - todomvc.mxml
      - sass
      - test
        - asconfig.json
        - pom.xml
        - todomvc.mxml.as3proj
```
Project Layout

This is the resulting project layout:
Open Visual Studio Code

Open VSCode and drag’n drop the project folder into @code to open the project.

Then **CMD+SHIFT+B** to open build tasks, and select “Build with Maven” to build the project.

You can run the starter project in the browser.
Laying out the interface

Creating the initial application visuals.
Main App
(App.mxml)

Root Application file:

- Jewel Application
- Crux Configuration
- initialView

Also we add styles (CSS) at this level.
Main Layout
(views/TodoListSection.mxml)

Let’s add some visual components and lay out our interface

- **Jewel View** is the main application view.
- Other components are html tags (Header, Section and Footer) to get the same look and feel as [todomvc.com](http://todomvc.com)
Jewel Group layout components at absolute positions as layers (one below the next):

- **Jewel TextInput** takes all available space and uses a **TextPrompt** bead.
- **Jewel ToggleButton** uses a **MaterialIcon** for the “arrow down” icon.
The Footer Bar
(views/TodoFooter.mxml)

We can choose between three different states, to show “all” items, just “active”, or “completed”.

In Royale you can declare different View States.

Then use “dot syntax” to specify if components appear in a particular state.
The Todo DataContainer
(views/TodoListSection.mxml)

The main todo list is just a Jewel DataContainer that will be filled with the items the user creates.

So we add the custom components recently created (The TodoHeader and TodoFooter) along with the main todo Jewel DataContainer to the main layout.
Setting up the MVC

MVC Design Pattern is the main organization in the example.
Creating the “M” and the “C”

We created the “V” (View) in the MVC design for our application. Now it’s time to create the “M” (Model) and the “C” (Controller).

From Wikipedia, the free encyclopedia

Model–view–controller (usually known as MVC) is a software design pattern[1] commonly used for developing user interfaces that divides the related program logic into three interconnected elements. This is done to separate internal representations of information from the ways information is presented to and accepted from the user.[2][3] This kind of pattern is used for designing the layout of the page.
The Model
(models/TodoModel.as)

The Todo Model stores global variables that the Controller updates and the View uses to keep the display current for the user.

We use [Bindable] metadata so we can update variables in the views “automagically”.

Collections are TodoVO instances.
Value Object
(vos/TodoVO.as)

TodoVO represents each piece of data (a todo item) the user creates in the application and adds it to the todo list.

We have a “label” for the description of the task and a Boolean “done” to check if it is complete.

It’s marked [Bindable] since we’ll be updating it through data binding.
Event Handling
(events/TodoEvent.as)

As a user interacts with a view, the app creates an Event object and passes it to the “C” (Controller).

The event instance is filled with the relevant piece of data (i.e: a TodoVO).

We set the event bubbles to true. Will come to this later.
The Controller
(controllers/TodoController.as)

Todo Controller holds all the global actions.

The user interacts with views. Views dispatch events that bubble. The controller registers events and updates the model so it can update views (using data binding most of the time).

We need to Inject the model to use here.
Glueing with Crux and its Beans

(config/Beans.mxml)

Crux lets us create simple classes and add them to a Crux BeanProvider to make them available in the rest of our code through injection.

package controllers
{
    public class TodoController
    {
        //example of injection where name of source is different to local name
        [Inject(source = "todoModel")]
        public var model:TodoModel;
    }
}
Adding a Todo Item

How to wire the use case of “adding an item to the list”
Adding a Todo Item (1)

After we set up all the pieces, we add some interaction to add a single item to the list.

We want the user to type any text in the TodoHeader.mxml’s TextInput (with localId="need"). Then hit Enter to add it to the list as a new todo item.
Adding a Todo Item (2)

1. [TodoEvent] Add The `event` action:

   ```
   public static const ADD_TODO_ITEM:String = "add_Todo_Item";
   ```

2. [TodoHeader] Add the input's `enter` event and create the event handler:

   ```
   <j:TextInput localId="need" ... enter="addItem(event)">
   ```
   ```
   // Signal todo item addition from main text box if text is not empty
   private function addItem(event:Event):void {
       if(event.target.text == "") return;
       var newTodo:TodoVO = new TodoVO(event.target.text);
       // Now we create and dispatch the new TodoEvent with the action and data we want to perform
       dispatchEvent(new TodoEvent(TodoEvent.ADD_TODO_ITEM, newTodo));
       event.target.text = "";
   }
   ```
Event Bubbling

TodoHeader listens for a TextInput Enter Event and calls an addItem callback handler when it hears one.

addItem creates the ADD_TODO_ITEM TodoEvent that will bubble up the display object hierarchy to Application.

Beans in a BeanProvider like the Controller use [EventHandler] to react to this event.

```javascript
[EventHandler(event="TodoEvent.ADD_TODO_ITEM", properties="todo")]
public function addTodoItem(todo:TodoVO):void {
    // We add the item to the ArrayList of items in the model
    model.allItems.addItem(todo);
    // This method perform various actions to update different interface parts
    updateInterface();
}
```

If your action requires to contact a server you can inject a **Crux ServiceHelper** and the `executeServiceCall` method.
Method to update the display

Here we run some code that updates the model variables. That causes a refresh in the views via data binding.

```javascript
protected function updateInterface():void {
    setListState();
    var item:String = model.activeItems.length == 1 ? " item ": " items ";
    model.itemsLeftLabel = model.activeItems.length + item + " left";
    model.clearCompletedVisibility = model.completedItems.length != 0;
    model.footerVisibility = model.allItems.length != 0;
    model.toogleAllVisibility = model.allItems.length != 0;
    model.toggleAllToCompletedState = model.activeItems.length == 0;
}
```

```javascript
// Sets the new state filter and refresh list to match the filter
protected function setListState():void {
    // setting to the same collection must cause refreshed too
    model.listItems = null;
    model.activeItems.refresh();
    model.completedItems.refresh();

    if(model.filterState == TodoModel.ALL_FILTER) {
        model.listItems = model.allItems;
    } else if(model.filterState == TodoModel.ACTIVE_FILTER) {
        model.listItems = model.activeItems as IArrayList;
    } else if(model.filterState == TodoModel.COMPLETED_FILTER) {
        model.listItems = model.completedItems as IArrayList;
    }
}
```
Trigger Data Binding

*views/ToDoListSection.mxml*

1. Inject the model.
2. Add the **ContainerDataBinding** bead to give binding functionality.
3. Fill Jewel **DataContainer**’s **dataProvider** with the **listItems** model’s variable.

Notice the data binding’s curly braces. As **listItems** changes, List updates to reflect the current state of the data model.
Let’s see how renderers work inside a list.
ItemRenderers

You can customize how each piece of information in a DataContainer displays.

We represent each TodoVO instance with a label and a green checkbox. We want to strike the label text when the item is complete and have an edit state.

* Configure renderer in CSS (and add style to component's className):

```html
.todo-list
  IItemRenderer: ClassReference("jewel.todomvc.renderers.TodoItemRenderer")
</j:DataContainer class="todo-list" .../>
```

* Configure renderer in MXML:

```xml
<j:DataContainer itemRenderer="jewel.todomvc.renderers.TodoItemRenderer" .../>
```
ItemRenderer

 renderItemers/TodoItemRenderer.mxml

The renderer lays out all visual controls in the item:

- A Checkbox to mark the item “done”.
- A Label for the item description.
- A TextInput when we are in edit mode.
- An IconButton to remove the item.

We use View States to change between normal and edit modes.

The renderer dispatches some TodoEvents that bubble up the display list.
Styling, Storage & Routing

Final touches to complete the example.
Styling Jewel Components

- **Jewel** uses **Jewel Theme** by default for application look & feel.
- At the framework level Jewel uses **SASS** ([https://sass-lang.com/](https://sass-lang.com/)) to organize styles. SASS is **optional** but recommended at the user level, for the same reason you use AS3 over JS.
- When your application is compiled, the Royale compiler generates CSS based on the styles in the framework and the user code.
- User CSS declarations take precedence over framework code, and the compiler removes duplicates.
Look & Feel
(sass/_global.sass)

Each Jewel Component has its own CSS declaration.

For example Jewel Button uses .jewel.button

We can change anything at the user level, to make components look the way we want.

Royale can define beads in CSS. Example: the TodoItemRenderer declaration.

// Button
.jewel.button
  margin: 3px
  padding: 3px 7px
  background: none
  border: 1px solid transparent
  box-shadow: none
  border-radius: 0.25rem
  color: #808080
  font-weight: normal
  text-shadow: none
  text-transform: initial

&:hover, &:hover:focus
  background: none
  border: 1px solid transparent

&:active, &:active:focus
  background: none
  border: 1px solid transparent
  box-shadow: none

&:focus
  background: 1px solid transparent
  box-shadow: none

// Data Container
.todo-list
  IItemRenderer: ClassReference("jewel.todomvc.renderers.TodoItemRenderer")
LocalStorage (AMF)

We can't save the user’s data yet. When you close the browser your data is lost. So we save the information locally with LocalStorage. Also we can simplify using the AMF (Action Message Format) protocol, so encoding and decoding are transparent to us.

Using JSON we need to add a method to encode each todo item, and a method to decode it.

You can use AMF with mx:RemoteObject (a Royale RPC class) and any of the AMF backend implementations (BlazeDS for Java, Fluorine for .NET, AMFPHP for PHP...)

LocalStorage (AMF)
The LocalStorageDelegate uses the AMFStorageBean and implements two methods to save and retrieve all items in the todo list.

To use AMF we add RemoteClass metadata to TodoVO so the AMF subsystems know how to encode/decode each item and save us from coding serialization methods:

```actionscript
package services {
    import org.apache.royale.crux.storage.AMFStorageBean;

    //LocalStorageDelegate stores data in Local browser storage
    public class LocalStorageDelegate implements ILocalStorageDelegate {
        [Inject(source="amfStorageBean", required="true")]
        public var storage:AMFStorageBean = null;

        //Retrieves the array of items
        public function getItemStore():Array {
            return storage.getValue("items", []) as Array;
        }

        //Saves the array of items
        public function setItemStore(items:Array):void {
            storage.setValue("items", items);
        }
    }
}
```

To make AMF work add ClassAliasBead in your application (check Main App slide).
LocalStorage (AMF)

Add LocalStorageDelegate and Crux AMFStorageBean to your Beans.mxml:

```xml
<services:LocalStorageDelegate id="localStorageDelegate"/>
<crux:AMFStorageBean id="amfStorageBean" name="todomvc" localPath="crux"/>
```

In Controller Inject the delegate:

```typescript
[Inject(source = "localStorageDelegate")]
public var delegate:ILocalStorageDelegate;
```

// Saves the actual data to the local storage via Local SharedObject
protected function saveDataToLocal():void {
  try {
    delegate.setItemStore(model.allItems.source);
  } catch (error:Error) {
    trace("You need to be online to store locally");
  }
}
Routing

Our App is SPA (Single Page Application). It all happens on the same page. But we can add Routing (a.k.a Deep Linking) to update the browser’s address bar as we change the app’s state.

In the Main View (TodoListSection.mxml) we add beads for Routing:

- **RouteToState** uses the **TodoFooter** component and its **View States** to change the route.
- **RouteTitleLookUp** updates the browser’s **window title** based on the current state.
Links

This presentation is about a real code example:

- **Full project source code**

- **Running Royale Application**
  - [https://royale.apache.org/todomvc-jewel/](https://royale.apache.org/todomvc-jewel/)
That's all! :)

![Image of a person looking upwards](image-url)
Thank you for your attention!

Questions?

royale.apache.org