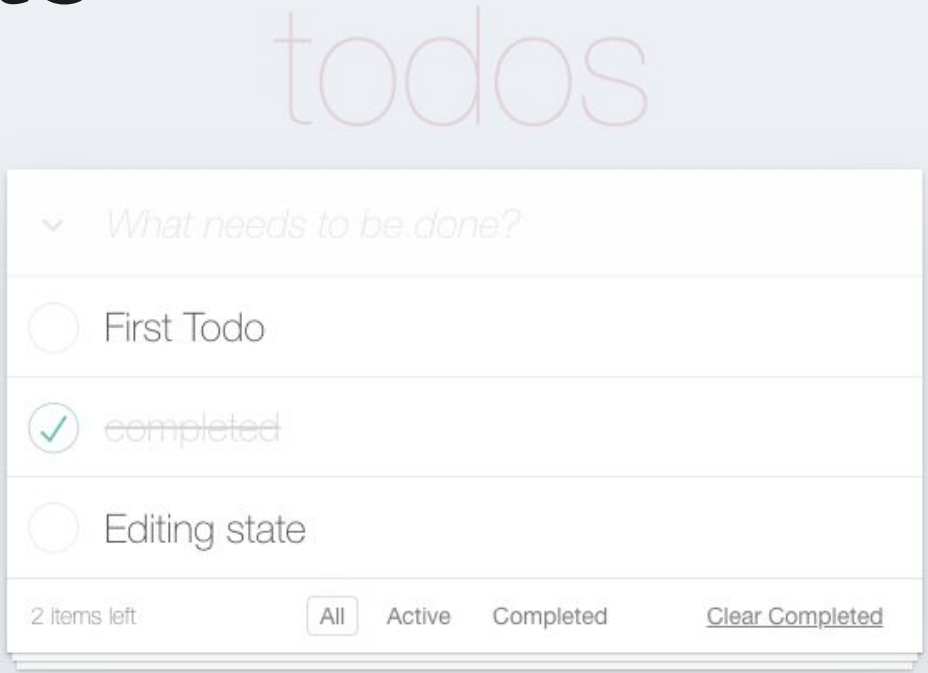
Apache ROYALE

THE APACHE SOFTWARE FOUNDATION http://www.apache.org/

Apache Royale

Starting from a blank file

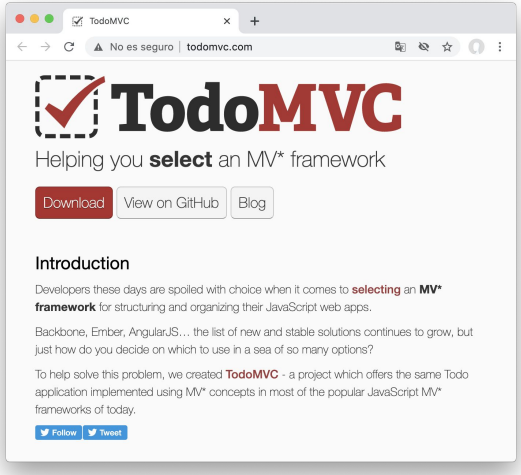


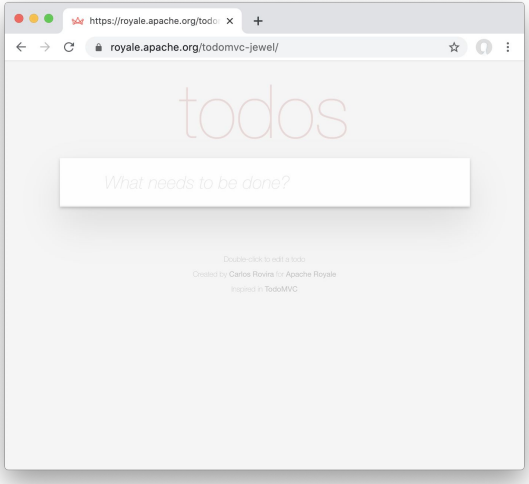
30.09.2020 20:15 CEST

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.

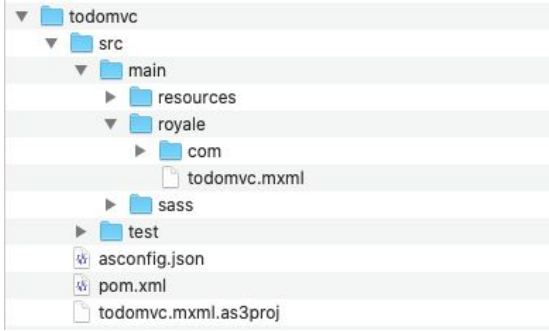
Create a project with a Maven Archetype

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
-DincludeCrux=true
```

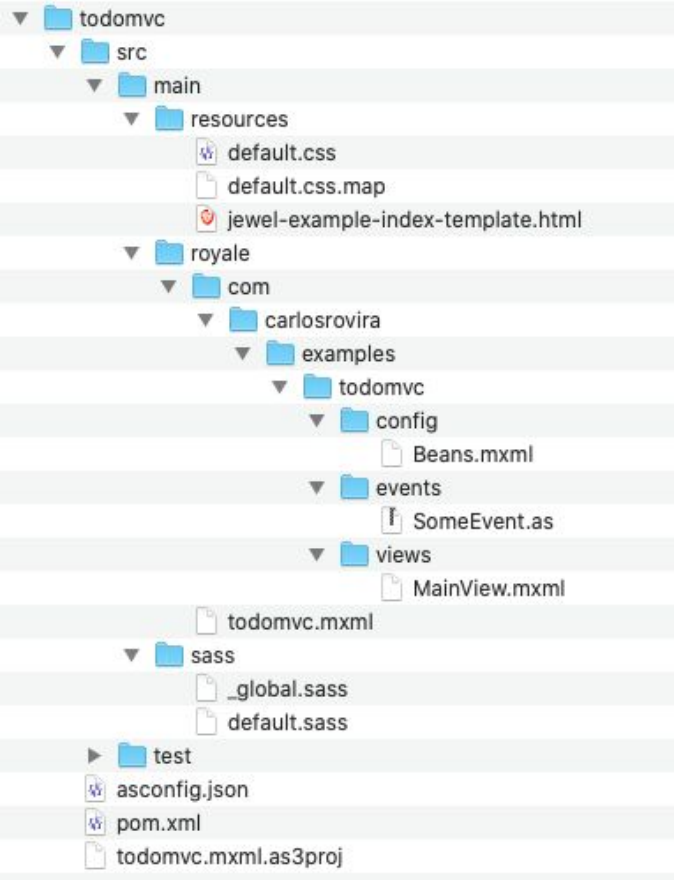
Example configuration:

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: :



Project Layout

This is the resulting project layout:

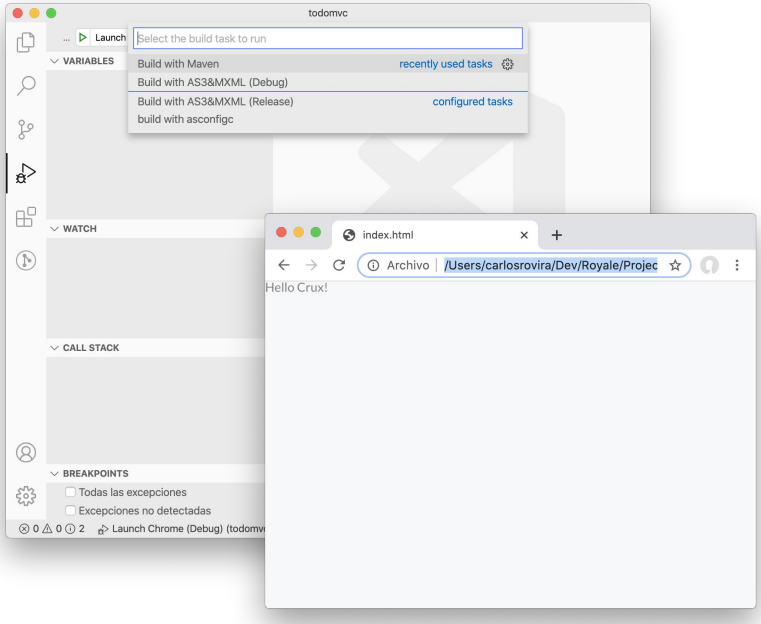


Open Visual Studio Code

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.

Apache ROYALE

THE APACHE SOFTWARE FOUNDATION http://www.apache.org/

Main App

(App.mxml)

Root Application file:

- Jewel **Application**
- Crux Configuration
- initialView

Also we add styles (CSS) at this level.

```xml
<?xml version="1.0" encoding="utf-8"?>
<j:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
               xmlns:j="library://ns.apache.org/royale/jewel"
               xmlns:js="library://ns.apache.org/royale/basic"
               xmlns:crux="library://ns.apache.org/royale/crux"
               xmlns:config="jewel.todomvc.config.*"
               xmlns:views="jewel.todomvc.views.*">

    <fx:Style source="../../main/resources/default.css"/>

    <j:valuesImpl>
        <js:SimpleCSSValuesImpl />
    </j:valuesImpl>

    <j:beads>
        <js:ClassAliasBead/>
        <crux:JSStageEvents packageExclusionFilter="_default_"/>
        <crux:Crux>
            <crux:beanProviders>
                <config:Beans/>
            </crux:beanProviders>
            <crux:config>
                <crux:CruxConfig
                    eventPackages="jewel.todomvc.events.*"
                    viewPackages="jewel.todomvc.views.*"/>
            </crux:config>
        </crux:Crux>
    </j:beads>

    <j:initialView>
        <views:TodoListSection/>
    </j:initialView>

</j:Application>
```

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

```xml
<?xml version="1.0" encoding="utf-8"?>
<j:View xmlns:fx="http://ns.adobe.com/mxml/2009"
   xmlns:j="library://ns.apache.org/royale/jewel"
   xmlns:html="library://ns.apache.org/royale/html">

   <html:Section className="todoapp">
       <!-- The Header Title -->
       <html:H1 text="todos"/>

       <html:Header>
           <!-- The todo input box -->
       </html:Header>

       <html:Section localId="main">
           <!-- The list of todo items -->
       </html:Section>

       <html:Footer className="footer">
           <!-- The bottom actions...  -->
       </html:Footer>
   </html:Section>

   <html:Footer className="info">
           <!-- Some footer text about the app  -->
   </html:Footer>
</j:View>
```

Main Input Box

(views/TodoHeader.mxml)

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.

```
<?xml version="1.0" encoding="utf-8"?>
<j:Group xmlns:fx="http://ns.adobe.com/mxml/2009"
   xmlns:j="library://ns.apache.org/royale/jewel"
   xmlns:js="library://ns.apache.org/royale/basic"
   height="65">

 <j:TextInput localId="need" width="100%" className="new-todo">
     <j:beads>
         <j:TextPrompt prompt="What needs to be done?"/>
     </j:beads>
   </j:TextInput>

 <j:ToggleButton localId="toogleAll"
     width="50" height="61" className="toggleAll">
     <j:icon>
         <js:MaterialIcon
text="{MaterialIconType.KEYBOARD_ARROW_DOWN}"/>
     </j:icon>
   </j:ToggleButton>

</j:Group>
```

Apache ROYALE

THE APACHE SOFTWARE FOUNDATION http://www.apache.org/

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.

```
<?xml version="1.0" encoding="utf-8"?>
<j:BarRow xmlns:fx="http://ns.adobe.com/mxml/2009"
   xmlns:j="library://ns.apache.org/royale/jewel"
   xmlns:js="library://ns.apache.org/royale/basic">

   <j:states>
       <js:State name="All"/>
       <js:State name="Active"/>
       <js:State name="Completed"/>
   </j:states>

   <j:BarSection width="15%">
       <j:Label/>
   </j:BarSection>

   <j:BarSection gap="3" itemsHorizontalAlign="itemsCenter">
       <j:ToggleButton selected.All="true" selected.Active="false"
selected.Completed="false" text="All"/>
       <j:ToggleButton selected.All="false" selected.Active="true"
selected.Completed="false" text="Active"/>
       <j:ToggleButton selected.All="false" selected.Active="false"
selected.Completed="true" text="Completed"/>
   </j:BarSection>

   <j:BarSection width="15%" gap="3" itemsHorizontalAlign="itemsRight">
       <j:Button text="Clear Completed" className="clearCompleted"/>
   </j:BarSection>
</j:BarRow>
```

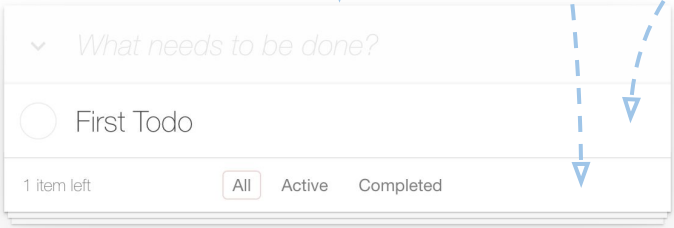
The Todo DataContainer

(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.

```
<j:View
    ...
    xmlns:views="views.*">

 <html:Section className="todoapp">
    <html:H1 text="todos"/>

    <html:Header>
       <views:TodoHeader/>
    </html:Header>

    <html:Section localId="main">
       <j:DataContainer width="100%" className="todo-list"/>
    </html:Section>

    <html:Footer className="footer">
       <views:TodoFooter currentState="All"/>
    </html:Footer>
```



Setting up the MVC

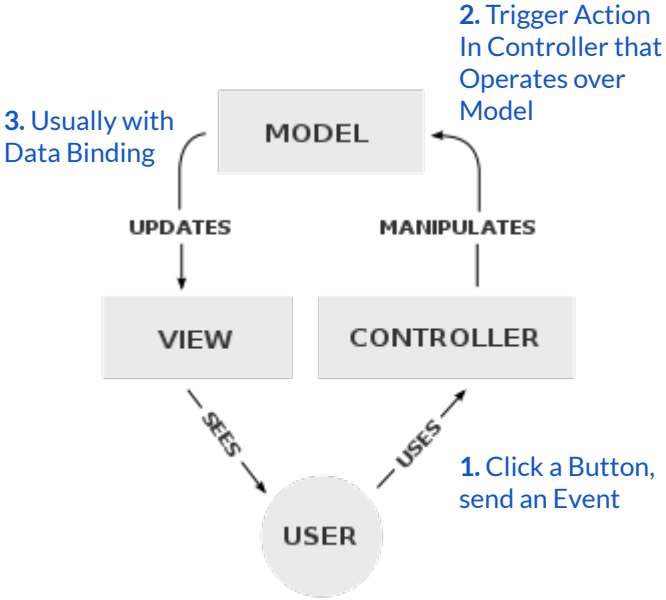
MVC Design Pattern is the main organization in the example.

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.

```
package models
{

   [Bindable]
   public class TodoModel
   {
       //the list of items binded to the todo list component
       public var listItems:IArrayLisy;

       // the real list with all items
       public var allItems:ArrayList;

       // the filtered list with active items
       public var activeItems:IArrayListView;

       // the filtered list with completed items
       public var completedItems:IArrayListView;

       // Stores the current filter for the list
       public var filterState:String = TodoModel.ALL_FILTER;

       ... and many other variables ...
   }
}
```

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.

```
package vos
{
    [Bindable]
    public class TodoVO
    {
        // label "todo" description
        public var label:String;

        // completion state (done/undone)
        public var done:Boolean;

        // constructor
        public function TodoVO(label:String)
        {
            this.label = label;
        }
    }
}
```

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.

```
package events
{
   import vos.TodoVO;

   import org.apache.royale.events.Event;

   public class TodoEvent extends Event
   {
       // Action constants
       // ... here we'll be adding constants

       // The todo to pass between layers
       public var todo:TodoVO;

       // This is a normal Royale event which will be dispatched
from a view instance. Only thing to note is we set 'bubbles' true,
so the event will bubble up the 'display' list, allowing Crux to
listen it.
       public function TodoEvent(type:String, todo:TodoVO = null)
       {
          super(type, true);

          this.todo = todo;
       }
   }
}
```

Apache ROYALE

THE APACHE SOFTWARE FOUNDATION http://www.apache.org/

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.

```
package controllers
{
  public class TodoController
  {

    /**
     *  Add the todo item to the list.
     *  Save data and refresh the list state
     */
    [EventHandler(event="TodoEvent.ADD_TODO_ITEM", properties="todo")]
    public function addTodoItem(todo:TodoVO):void {
      model.allItems.addItem(todo);

      saveDataToLocal();
      updateInterface();
    }

  }
}
```

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.

```
<?xml version="1.0" encoding="utf-8"?>
<crux:BeanProvider
   xmlns:fx="http://ns.adobe.com/mxml/2009"
   xmlns:crux="library://ns.apache.org/royale/crux"
   xmlns:models="models.*"
   xmlns:controller="controllers.*">

   <models:TodoModel id="todoModel"/>
   <controller:TodoController id="todoController"/>

</crux:BeanProvider>
```

(controllers/TodoController.as)

Here we inject the model bean in the controller bean using metadata [Inject]

```
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:

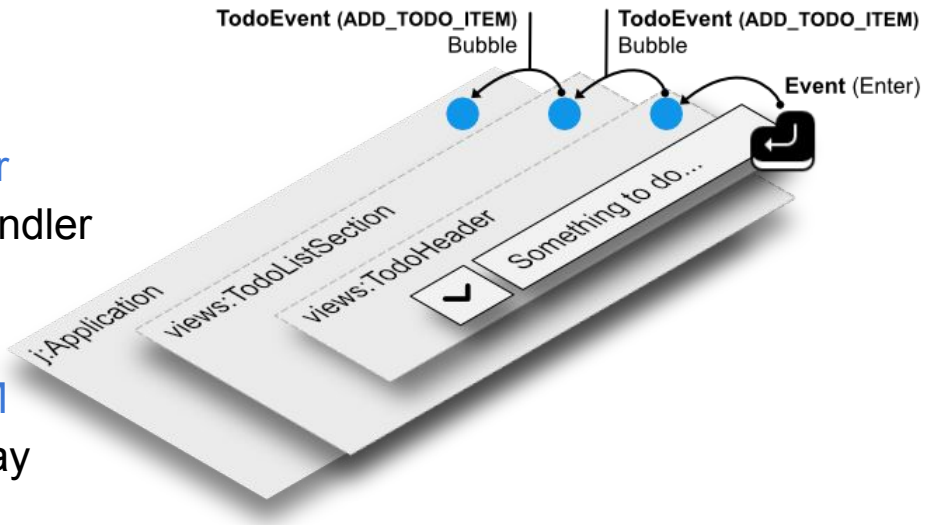
```
<j:TextInput localId="need" ... enter="addItem(event)">

//  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.



Adding a Todo Item (3)

3. [TodoController] Add the event handler with the [EventHandler] metadata.

```
[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**(call, resultHandler, faultHanlder) method.

Method to update the display

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

```
//  Update the interface accordingly
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;
}
```

```
// 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;
  }
}
```

Apache ROYALE

THE APACHE SOFTWARE FOUNDATION http://www.apache.org/

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.

```
<fx:Script>
    <![CDATA[
        import models.TodoModel;

        [Inject]
        [Bindable]
        public var todoModel:TodoModel;
    ]]>
</fx:Script>

<j:beads>
    <js:ContainerDataBinding/>
</j:beads>

<html:Section className="todoapp">
    ...
    <html:Section localId="main">
        <j:DataContainer localId="todolist" width="100%"
            labelField="label" className="todo-list"
            dataProvider="{todoModel.listItems}"/>
    </html:Section>
    ...
</html:Section>
```

Notice the data binding's curly braces. As **listItems** changes, List updates to reflect the current state of the data model.

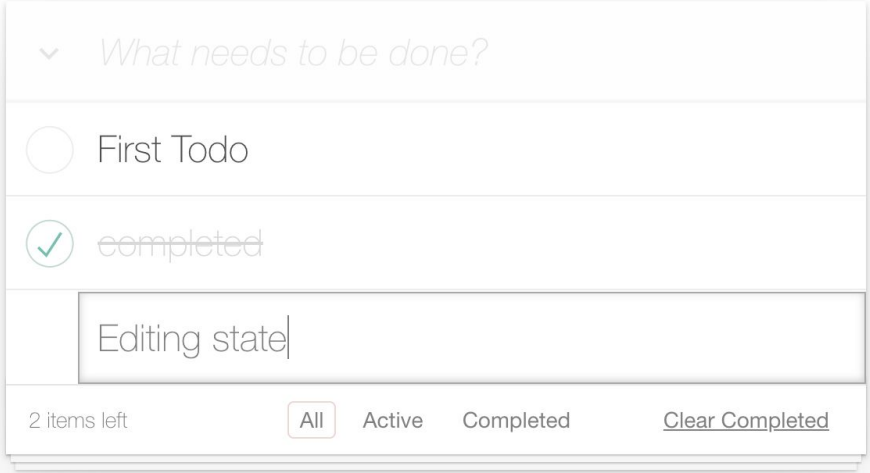
Drawing each list Item

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

* Configure renderer in MXML:
<j:DataContainer itemRenderer="jewel.todomvc.renderers.TodoItemRenderer" .../>
```

ItemRenderer

(renderers/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.

```
<j:ListItemRenderer xmlns:fx="http://ns.adobe.com/mxml/2009"
   xmlns:j="library://ns.apache.org/royale/jewel"
   xmlns:js="library://ns.apache.org/royale/basic"
   width="100%" minHeight="60">

   <j:states>
       <js:State name="normal"/>
       <js:State name="editing"/>
   </j:states>

   <j:beads>
       <js:SimpleStatesImpl/>
       <js:ItemRendererDataBinding />
       <j:HorizontalLayout itemsVerticalAlign="itemsCenter"/>
       <js:Paddings padding="0"/>
   </j:beads>

   <j:CheckBox .../>
   <j:Label localId="description" .../>
   <j:TextInput localId="editfield" .../>
   <j:IconButton localId="destroy_btn" .../>
</j:ListItemRenderer>
```

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/) 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.

Apache ROYALE

THE APACHE SOFTWARE FOUNDATION http://www.apache.org/

Look & Feel

(sass/_global.sass)

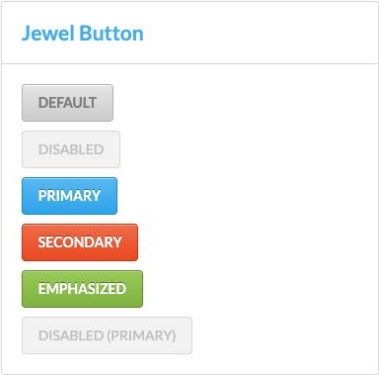
Each Jewel Component has its own **CSS** declaration.

For example **Jewel Button** uses **.jewel.button**

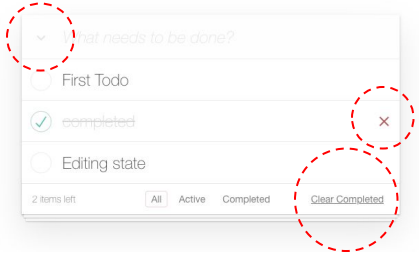
We can change anything at the user level, to make components look the way we want.

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
       border: 1px solid transparent
       box-shadow: none
```



Jewel Button Default Look & Feel



Styled Look & Feel for Jewel Buttons in TodoMVC example

```
// 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...)

Apache ROYALE

THE APACHE SOFTWARE FOUNDATION http://www.apache.org/

LocalStorageDelegate

(services/LocalStorageDelegate.as)

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:

```
[Bindable]
[RemoteClass(alias="vos.TodoVO")]
public class TodoVO
```

```
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 ot items
      public function getItemStore():Array
      {
          return storage.getValue("items", []) as Array;
      }

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

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

```
<j:beads>
      <js:ClassAliasBead/>
...
```

LocalStorage (AMF)

Add LocalStorageDelegate and Crux AMFStorageBean to your Beans.mxml:

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

In Controller Inject the delegate:

```
[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

```
<j:beads>
    ...
    <js:Router>
        <js:RouteToState component="{footer}"/>
        <js:RouteTitleLookup lookup="{getTitleLookup()}"/>
    </js:Router>
</j:beads>
```

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
  - https://github.com/apache/royale-asjs/tree/develop/examples/crux/todomvc-jewel-crux
- Running Royale Application
  - https://royale.apache.org/todomvc-jewel/

That's all! :)

Thank you for your attention!

Questions?

royale.apache.org