Getting Started with ECML

Simplest way to get started is to use one of the sample story or worksheets to get started. These can be downloaded from the following URLs. Once you have downloaded the files, unzip it into a local folder, and open index.html in your browser.

1. Sample Worksheet - Addition with Grouping
2. Sample Story - Annual Haircut Day

Note: Browser security settings may prevent the renderer from reading the markup XML file. Default browser settings for Firefox usually permit this, whereas Chrome prevents the renderer to read the markup XML.

If the index.html does not open in either of the browsers, then you should install a webserver like Apache and then access index.html using http:// URL instead of opening the file directly from the file system.

For a complete reference of the markup, visit ECML Content Document Spec v1.0

Working with the Markup

Open the index.ecml in any text editor and start editing. Important aspects of the markup are explained below. For a full reference, see the references section.

- Declaring the assets
- Building the stages
- Extending stages
- Defining events and actions
- Navigating between stages
- Playing audio automatically on stages
- Using audio sprites
- Building interactive hotspots
- Animating objects (path animations)
- Show/Hide objects (e.g. speech bubble)

- Weaving external content and items
- Using item templates
- Accepting input and evaluating answer
- Multiple choice and match the following questions
- Defining events and actions on options
- Using data templates and stage iteration on data
- Using sprite sheets
- Links to external pages
- Audio recording and processing using Sensibol
- Audio recording and playback last recorded audio
- Using Images
- Using Text
- Using Shape
- Using Scribble
- Delay an action
- Using HTML
- Using Video
- Using Grid
- Show and Hide HTML Elements
- Teacher Instructions

Declaring the assets

Every asset (image or sound file) that is used in the markup must be listed in the manifest section of the markup XML. The element has three attributes.

- "id" that you assign to the asset which you will use subsequently.
- "src" identifies the physical asset file, the path of the file is relative to the assets folder.
- "type" identifies the type of asset - can be either image, audio or json (for data)

```
<theme>
    <manifest>
        <media id="razor" src="razor.png" type="image"/>
        <media id="clock_sound" src="sounds/clock_sound.mp3" type="sound" />
        <media id="story_data" src="data_english.json" type="json"/>
```

Building the stages

A story or worksheet consists of multiple pages. You add pages to the story by defining them as stages. Each stage has a unique id and the dimensions (x, y, w, h). Typically, you would have cover pages in a story, or splash stages in a worksheet, followed by individual pages in the story/worksheet and finally an end page. Conceptually, it is identical to a powerpoint presentation where the story consists of individual slides.

```
<stage id="splash" x="0" y="0" w="100" h="100">
```

The first stage of your worksheet/story should be specified in the `<theme>` tag using the attribute `"startStage=<stage_id>"` - the renderer starts the story from this stage.

Tips:

- A stage need not be full page - part of a page can also be defined as a stage. In such a case, you can use the x,y,w,h attributes to constrain the bounding rectangle of the stage
- When you are developing the story and say developing page-3 of the story, when you edit the markup and refresh the browser, it will start again from the page-1. To start at the current page, you can make this page as the start page. When done, you can make the page-1 again as the start page. You should have only one start page.

Extending stages

Common UI elements that are required to be displayed in all the stages of a story can be defined in a common base stage and other stages can extend the base stage.

```
<!-- Create stages that can be extended by other stages -->
  <stage id="baseStage" preload="true">
    <image asset="logo" x="5" y="5" w="15" h="10"/>
  </stage>
  <stage id="splash" extends="baseStage">
<!-- Logo image defined in the baseStage will be added to this stage -->
    ...
  </stage>
```

Defining events and actions

Events can be broadly categorised into two types -

- DOM Events: Events that are triggered by interactions with the DOM elements, like click, touch, mousedown, mouseover, etc.
- App Events: Events that are specific to the application, like enter, exit, eval, correct_answer, wrong_answer, etc

Events can be defined in the markup to listen on the DOM events. Also, events can be created for all types of app events. For each and every event, there can be multiple actions configured that need to happen when the event occurs.

To listen to a click event on an image:

```
<image asset="next">
  <event type="click">
    <action type="command" command="play" asset="sound1"/>
  </event>
</image>
```

To define a custom app event:

```
<events>
  <!-- App event 'enter' that is triggered on stage enter -->
  <event type="enter">
    <action type="command" command="play" asset="sound1"/>
  </event>
  <!-- Custom event 'correct_answer' that can be triggered by other actions -->
  <event type="correct_answer">
    <action type="command" command="play" asset="applaud"/>
  </event>
</events>
```

Two types of actions can be defined in the markup:

- Command: A set of pre-defined commands can be used to perform various actions.
- Animation: An animation action will be triggered for this type of action.

- Note: We are deprecating the animation type. It is also a command from now. Please don't use type="animation".

List of available commands:

- `play`: To play an audio or a audio sprite
- `pause`: To pause an audio
- `stop`: To stop playing an audio
- `togglePlay`: To toggle playing of an audio
- `show`: To show a hidden graphic object on the canvas
- `hide`: To hide a graphic object
- `toggleShow`: To toggle display of a graphic object
- `transitionTo`: To transition to another stage
- `event`: To trigger an app event
- `toggleShadow`: To toggle shadow of a graphic object
- `windowEvent`: To change the window location
- `eval`: To evaluate items in a stage
- `reload`: To reload the current stage
- `erase`: To erase the scribble pad.
- `external`: To open an external URL in browser.
- `refresh`: To refresh a display object.
- `set`: To set a param with scope.
- `startRecord`: To start audio recording.
- `stopRecord`: To stop audio recording.
- `blur`: To blur a display object.
- `unblur`: To unblur a display object.
- `custom`: To invoke a method of a plugin object.
- `showhtmlelements`: To show all the HTML elements/HTML Layer in a stage.
- `hidehtmlelements`: To hide all the HTML elements/HTML Layer in a stage.
- `animate`: An animation action will be triggered for this type of action.

Play actions in sequence

In ECML, the actions are part of an event. All the actions part of an event will starts to execute at the same time when the event occurs. We have attributes `after` and `with` on action tag to decide when to play these actions.

- `after` - Identifier of the action. Current action will be invoked after given action completes.

- `with` - Identifier of the action. Current action will be invoked along with given action.

With this we can define multiple levels of sequencing and parallel actions. Below is the sample.

```
 |onClick of a Display Object
  | |- action1
  | |   |-action2 (after action1)  -> (action2, action3 are in parallel)
  | |   |-action3 (with action2)   ->
  | |       |-action6 (after action3)
  | |- action4
  | |- action5
  |
```

Below is the sample ECML structure:

```
<events>
    <event type="click">
        <action id="action1"/>
        <action id="action2" after="action1"/>
        <action id="action3" with="action2"/>
        <action id="action6" after="action3"/>
        <action id="action4"/>
        <action id="action5"/>
    </event>
</events>
```

Navigating between stages

To navigate between stages, you have to define the "previous" and “next” events and define the stages to link to. The previous and next events are fired by clicking on back and next buttons that should be added to stages to control the flow. For the back and next events, you can specify the transition effects as shown in the examples.

- "transitionTo" - target stage to transition to (for previous and next stages)
- "effect" - type of transition effect (scroll, fadeIn, fadeOut)
- "direction" - from which the effect starts (e.g. left, right, top, down)
- "ease" - easing effect on the transition (acceleration, deceleration etc)

- "duration" - duration in milliseconds to complete the transition.

Note: In future the navigation controls will be removed from the markup so it is recommended to keep them distinct from the rest of the assets.

```
<image asset="next">
  <event type="click">
    <action type="command" command="transitionTo" asset="theme" value="stage2" effect="scroll" direction="top" ease="linear" duration="500"/>
  </event>
</image>

<image asset="previous">
  <event type="click">
    <action type="command" command="transitionTo" asset="theme" value="splash" effect="scroll" direction="top" ease="linear" duration="500" transitionType="previous"/>
  </event>
</image>
```

Playing audio automatically on stages

To play audio automatically when a stage is rendered (e.g. for stories), you have to bind the audio asset to the "enter" event. To stop playing the audio when the stage is transitioned (either back and next), you bind “exit” event.

- "audio" - mark the event handler as an audio handler by setting the value true
- "type" - play or stop or toggle audio (e.g. depending upon entry or exit into the stage)
- "loop" - number of iterations to play the audio (e.g. you can play it in a loop), but you must attach a stop event on exit of the stage.

```
<stage is="stage1" x="0" y="0" w="100" h="100"/>
  <audio type="scene1_sound"/>
  <events>
   <event type="enter">
     <action type="command" command="play" asset="sound1"/>
   </event>
```

```
 </events>
</stage>
```

Using audio sprites

Audio sprites can be used to optimise the number of audio files used in a story or a worksheet. Multiple audio files can be merged into a single audio file and each of the audio files can be defined as a audiosprite asset in the manifest.

```
<!-- Audio file spaceship-sprite.mp3 has multiple parts that can be used in separate actions
-->
<media id="spaceship" uri="spaceship-sprite.mp3" type="audiosprite">
  <data>
    <audioSprite id="sound1" startTime="0" duration="2821.1"/>
    <audioSprite id="sound2" startTime="3000" duration="1728"/>
  </data>
</media>

<!-- On enter, play the sound1 which is from 0 to 2821.2ms in the audio file -->
<event type="enter">
  <action type="command" command="play" asset="sound1"/>
</event>

<!-- On exit, play the sound2 which is from 3000 to 4728ms in the audio file -->
<event type="exit">
  <action type="command" command="play" asset="sound2"/>
</event>
```

Building interactive hotspots

To build interactivity into the app, we have to respond to events such as on click of an image or a shape. You can create an interactive hotspot for an image (e.g. an icon) or a shape (e.g. a hidden rectangle on top of the image of water in the background). You place the image or shape at the desired x,y,h,w location in the canvas and then define the event to fire on click. In the

handler for that event, you can define the action. Sample below shows how to play and stop (toggle) an audio when clicking on an image or shape.

```
   <image asset="icon_sound" x="92" y="80" w="6" h="10"/>
<!-- Create a hotspot on the above image to play a sound on click -->
      <shape type="rect" x="87" y="75" w="13" h="20" hitArea="true">
            <event type="click">
       <action type="command" command="play" asset="sound2"/>
     </event>
      </shape>
```

Animating objects (path animations)

Another type of interactivity is to animate objects on the screen, typically in response to a touch/click. For example, to move a shape, you define the event on an image (the trigger need not be same as the asset that is actually animated) and define a "tween" path to animate the object. In this example, the image of the character Sringeri walks on a path as defined between the x and y values. The duration and easing of each element of the path is controlled as shown:

```
   <image asset="sringeri" x="11.5" y="19" w="20" h="36" id="sringeri">
  <event type="click">
    <action type="animation">
        <tween id="sringeri_walking">
              <to ease="linear" duration="500"><![CDATA[{"x":20,"y":20}]]></to>
              <to ease="quadOut" duration="2000"><![CDATA[{"x":55,"y":0}]]></to>
               <to ease="linear" duration="1"><![CDATA[{"x":75,"y":0, "scaleX": -1}]]></to>
              <to ease="linear" duration="2000"><![CDATA[{"x":40,"y":55}]]></to>
              <to ease="linear" duration="1"><![CDATA[{"x":18,"y": 55, "scaleX": 1}]]></to>
              <to ease="linear" duration="2000"><![CDATA[{"x":57,"y":55}]]></to>
        </tween>
    </action>
  </event>
</image>
```

Show/Hide objects (e.g. speech bubble)

Another type of interactivity is to show/hide some part of the assets - e.g. a speech bubble or a message. You can do this by defining an onclick event on the shape/image and binding it to a container event as shown below. In this example, the group "razorInfo" contains images and text that are shown when the user clicks/touches on the hidden rectangle shape.

Notice that you can fire multiple events (show the speech bubble and play an audio) by raising multiple events using a comma-separated list.

- "container" - mark the event as a container event (as against audio playback events)
- "type" - show to make the asset visible, hide to make it hidden, or toggle

```
<image asset="razor">
  <event type="click">
    <action type="command" command="event" asset="stage" value="showRazorInfo"/>
  </event>
</image>
...
<g id="razorInfo">
  ...
  <event type="click">
    <action type="command" command="event" asset="stage" value="hideRazorInfo"/>
  </event>
</image>

<events>
  <event type="showRazorInfo">
    <action type="command" command="show" asset="razorInfo"/>
    <action type="command" command="play" asset="razorInfoSound"/>
  </event>
  <event type="hideRazorInfo">
    <action type="command" command="hide" asset="razorInfo"/>
    <action type="command" command="stop" asset="razorInfoSound"/>
  </event>
</events>
```

Weaving external content and items

Controllers can be used to load data to be rendered on the canvas. Controllers help to separate data from the layout. There are two types of controllers available:

- Data Controller: For the textual content displayed in the story or worksheet
- Items Controller: For assessment items in the story or worksheet

Data Controller loads the JSON file from the "data" folder in the zip file. The data file is a JSON file in the following format:

```
{
  "item_title": "ADDITION - By Grouping",
  "title": "Noah's Ark...",
  "description": "Add the things in pictures to find out the total number of beings that Noah is carrying in his spaceship to Mars."
}
```

A data controller can be defined in the markup using tag:

```
<!-- ‘name’ is used as a reference the controller, ‘type’ is ‘data’ for Data Controllers and ‘id’ is the JSON file name -->
<controller name="ws_data" type="data" id="ws_data"/>

<!-- the controller can be used to provide data for other graphic objects -->
<text x="10" y="10" w="10" h="10" model="ws_data.item_title"/>
```

Item Controller loads the JSON file from the "items" folder in the zip file. The data file is a JSON file as per the EkStep assessment item definition. ItemController provides a two-way binding for the items data, i.e. user’s input to an assessment item is updated back in the controller’s model. Item Controller uses this data to evaluate the user’s response to an item.

```
<!-- 'name' is used as a reference the controller, 'type' is items for Item Controllers and 'id' is the JSON file name -->
<controller name="ws1" type="items" id="addition"/>

<!-- Item Controller can be used to repeat the stage multiple times each time presenting a different item from the items JSON file -->
<stage id="items" iterate="ws1" var="item">
```

```
  <placeholder model="item.param1" x="5" y="15" w="25" h="25">
  <placeholder model="item.param2" x="37" y="15" w="25" h="25">
  <placeholder model="item.param3" x="2" y="66" w="80" h="15">
  <input model="item.ans1" type="number" class="inputText" x="14" y="57" w="7" h="7">
  <input model="item.ans2" type="number" class="inputText" x="43" y="57" w="7" h="7">
  <input model="item.ans3" type="number" class="inputText" x="28" y="90" w="7" h="7">
</stage>
```

Item or Data controller loads the inline JSON data (in CDATA format).

```
<!-- ‘name’ is used as a reference the controller, ‘type’ is ‘data’ for Data Controllers and ‘id’ is the identifier -->
<controller id="story_data" name="storyData" type="data">
    <![CDATA[{
    "model" :{
        "ts_textpg_align" : "Text Plugin - Align",
        "ts_textpg_wrap" : "Text Plugin - Wrap",
        "ts_z_index": "Testing z-index rendering",
        "ts_hybrig_rendering": "Hybrid Rendering",
        "ts_font_rendering": "Mulpitule Font Rendering",
        "ts_external_launch" : "External Launch",
        "ts_cond_render_app": "Conditional Rendering - App level",
        "summary_heading" : "It works! See you soon. :)"
    }
}]]>
</controller>
```

Using item templates

You can define item templates in the markup to play multiple assessment items of the same type. Also, the template binding to the stage can happen dynamically by using the item metadata. The items data file may contain multiple items that can be rendered using various different templates.

```
// Item metadata from items JSON file
{
    "identifier": "ws1_set_2_1",
```

```
    "template": "vertical_group_addition",
    "model": {...},
    ...
},
{
    "identifier": "ws1_set_2_2",
    "template": "horizantal_group_addition",
    "model": {...},
    ...
}

<!-- templates can be defined in the markup -->
<template id="vertical_group_addition">
  <placeholder model="item.param1" x="5" y="15" w="25" h="25">
  <placeholder model="item.param2" x="37" y="15" w="25" h="25">
  <placeholder model="item.param3" x="2" y="66" w="80" h="15">
  <input model="item.ans1" type="number" class="inputText" x="14" y="57" w="7" h="7">
  <input model="item.ans2" type="number" class="inputText" x="43" y="57" w="7" h="7">
  <input model="item.ans3" type="number" class="inputText" x="28" y="90" w="7" h="7">
  <image asset="separator_horizontal" x="2" y="65" w="58" h="1">
</template>
<template id="horizantal_group_addition">
  <placeholder model="item.param1" x="5" y="15" w="25" h="25">
  <placeholder model="item.param2" x="37" y="15" w="25" h="25">
  <placeholder model="item.param3" x="67" y="15" w="30" h="25">
  <input model="item.ans1" type="number" class="inputText" x="5" y="40" w="7" h="7">
  <input model="item.ans2" type="number" class="inputText" x="37" y="40" w="7" h="7">
  <input model="item.ans3" type="number" class="inputText" x="67" y="40" w="7" h="7">
  <image asset="separator_horizontal" x="2" y="65" w="58" h="1">
</template>

<!-- Stage can pick the template dynamically for the current item being rendered  -->
<stage id="items" iterate="ws1" var="item">
<!-- Uses the template ‘vertical_group_addition’ for first item and ‘horizantal_group_addition’
for the second item  -->
    <embed template="item" var-item="item" var-data="data">
</stage>
```

Accepting input and evaluating answer

Accepting child input and evaluating the answer involves three parts to it:

1. Displaying input fields to accept keyboard input
2. Raising an eval event to trigger evaluation
3. Defining events to handle correct or incorrect answer

To define the input fields, use the `<input>` element. It is important to bind the input element to a parameter. Example above binds three input fields to `ans1`, `ans2` and `ans3` of the item model. The evaluation system matches the values against the expected values for these three parameters in the items.json file (see the example worksheet). Typically there would be a button/image to initiate evaluation. Bind an eval event to that image.

When the event is fired, you need to declare the event type as eval and identify the IDs of the input fields that need to be passed to the evaluation system. Define event handles for success and failure (for correct or incorrect answers). In the success/failure event handlers, you can show/hide cues for the same, play audio or run an animation as described previously.

```
<image asset="validate">
  <event type="click">
 <action type="command" command="eval" asset="items" success="correct_answer"
failure="correct_answer"/>
  </event>
</image>

<events>
  <event type="correct_answer">
    <action type="command" command="show" asset="correctAnsPopup"/>
  </event>
  <event type="wrong_answer">
    <action type="command" command="hide" asset="wrongAnsPopup"/>
  </event>
</events>
```

Multiple Choice and Match the Following questions

The tags `<mcq>`, `<mtf>`, `<options>` and `<option>` can be used to define templates for MCQ and MTF items. The data to these objects is also provided by an Item Controller, as shown for Fill in the Blanks type questions in the above sections.

- "showImmediateFeedback" - To show or hide feedback popup showing after evaluation. default is true.
- "shuffle" - To shuffle the options. default is false.

Items data for MCQ and MTF questions should be in the following JSON format:

```
// Item metadata from items JSON file
{
  "identifier": "ws1_set_2_1",
  "template": "mcq_template_1",
  "type": "mcq",
  "max_score": 4,
  "showImmediateFeedback": true,
  "shuffle": false,
  "title": "Find the Barber.",
  "options": [
    {"value": {"type": "image", "asset": "carpenter_img"}},
    {"value": {"type": "image", "asset": "barber_img"}, "answer": true},
    {"value": {"type": "image", "asset": "tailor_img"}},
    {"value": {"type": "image", "asset": "wife_img"}}
  ],
  "model" : {
    "title_audio" : {"asset": "learning10_sound", "type": "audio"}
  },
  "hints": [
    {"asset": "learning11_sound", "type": "audio"},
    {"asset": "Barber has Scissors to Cut hair.", "type": "text"}
  ]
  ...
},
{
```

```
  "identifier": "ws1_set_2_2",
  "template": "mtf_template_1",
  "type": "mtf",
  "max_score": 4,
  "showImmediateFeedback": true,
  "shuffle": false,
  "title": "Match the tools to the people.",
  "lhs_options": [
    {"value": {"type": "image", "asset": "carpenter_img"}, "index": 0},
    {"value": {"type": "image", "asset": "barber_img"}, "index": 1},
    {"value": {"type": "image", "asset": "tailor_img"}, "index": 2},
    {"value": {"type": "image", "asset": "wife_img"}, "index": 3}
  ],
  "rhs_options": [
    {"value": {"type": "image", "asset": "knife"}, "answer": 3},
    {"value": {"type": "image", "asset": "razor"}, "answer": 1},
    {"value": {"type": "image", "asset": "scissors"}, "answer": 2},
    {"value": {"type": "image", "asset": "saw"}, "answer": 0}
  ],
  ...
}
```

You can define the templates to render these items using mcq, mtf, options and option plugins:

```
<template id="mcq_template_1">
<!-- Create a MCQ widget backed by the model from item controller -->
  <mcq model="item" multi_select="false">

<!-- Layout the MCQ options in a table layout.
  Use x,y,w,h for the table. Specify the number of columns in cols attribute.
  marginX and marginY are used for specifying the margins between columns and rows.
  padX and padY are used to specify the padding to be given in each cell.
  The ‘options’ attribute is the options attribute name in the item JSON
-->
    <options layout="table" x="20" y="15" w="70" h="85" cols="2" marginX="10" marginY="5"
padX="5" padY="5" options="options" />
  </mcq>
```

```
</template>

<template id="mtf_template_1">
<!-- Create a MTF widget backed by the model from item controller -->
  <mtf model="item">

<!-- Layout the MTF left side options in a table layout.
  snapX and snapY are used to specify the location where a matched item should be snapped to
when matched with the current option
-->
    <options layout="table" x="15" y="15" w="75" h="85" cols="2" marginX="15" marginY="5"
snapX="45" snapY="35" options="lhs_options" />

<!-- Layout the MTF right side options individually using <option> tag. -->
    <option x="45" y="20" w="15" h="20" padX="20" padY="10" option="rhs_options[0]" />
    <option x="45" y="37" w="15" h="20" padX="20" padY="10" option="rhs_options[1]" />
    <option x="45" y="59" w="15" h="20" padX="20" padY="10" option="rhs_options[2]" />
    <option x="45" y="80" w="15" h="20" padX="20" padY="10" option="rhs_options[3]" />

  </mtf>
</template>

<!-- Stage can pick the template dynamically for the current item being rendered  -->
<stage id="quiz" iterate="assessment" var="item">
    <embed template="item" var-item="item" />
</stage>
```

You can drop multiple object(RHS) into a single option(LHS) by enabling multiple equals true.

```
   <option h="24.5" multiple="true" option="lhs_options[0]" snapX="12" snapY="12" w="15" x="65"
y="15"/>
    <options cols="2" h="37" layout="table" marginX="15" marginY="5" multiple="true"
options="lhs_options" snapX="45" snapY="35" w="51" x="60" y="18"/>
```

See the example story for sample usages of MCQ and MTF items.

Defining events and actions on options

On clicking of an option, playing an audio or show/hide an element.

On drag/drop of an option, play and pause of an audio clip.

This we can do by registering options with events. Below are the sample ECML for the same.

Playing audio on click of any one of MCQ option:

```
   <options layout="table" x="20" y="15" w="70" h="85" cols="2" marginX="10" marginY="5"
options="options">
        <events>
            <event type="click">
                <action type="command" command="play" asset="scene19_sound" loop="1" />
            </event>
        </events>
    </options>
```

Play audio on drag and stop audio on drop of the MTF option:

```
   <option x="70" y="71" w="12" h="22" option="rhs_options[2]">
        <events>
            <event type="mousedown">
                <action type="command" command="play" asset="drag_audio" audio="true" loop="1"
/>
            </event>
            <event type="pressup">
                <action type="command" command="stop" asset="drag_audio" audio="true" loop="1"
/>
            </event>
        </events>
    </option>
```

Using data templates and stage iteration on data

You can define data templates in the markup to play multiple data objects of same structure and layout. It works same as how item templetes work. The data file may contain multiple data objects that can be rendered using various different templates.

```
   // metadata from data JSON file
[{
    "title": "Page 1 - of the data iterator.",
    "description": "Description of Page 1.",
    "template": "dataTemplate"
}, {
    "title": "Page 2 - of the data iterator.",
    "description": "Description of Page 2.",
    "template": "dataTemplate1"
}, {
    "title": "Page 3 - of the data iterator.",
    "description": "Description of Page 3.",
    "template": "dataTemplate"
}]
```

```
   <controller name="testData2" type="data" id="testdata2"/>
    <!-- templates can be defined in the markup -->
    <template id="dataTemplate">
        <text model="data.title" x="9" y="7" w="86" h="10" color="green"  font="Georgia"
fontsize="100"/>
        <text model="data.description" x="9" y="20" w="86" h="50" font="Georgia"
fontsize="42"/>
    </template>
    <template id="dataTemplate1">
        <text model="data.title" x="9" y="7" w="86" h="10" color="red" font="Georgia"
fontsize="100"/>
        <text model="data.description" x="9" y="20" w="86" h="50" font="Georgia"
fontsize="42"/>
    </template>
    <!-- Stage can pick the template dynamically for the current data being rendered  -->
    <stage id="dataRepeatStage" x="0" y="0" w="100" h="100" iterate="testData2" var="data"
extends="baseStage">
        <param name="next" value="finished" />
```

```
        <param name="previous" value="audioRecording" />
        <embed template="data" var-data="data"/>
    </stage>
```

Using sprite sheets

A sprite sheet is a series of images (usually animation frames) combined into a larger image. You place the sprite image at the desired x,y,h,w location in the canvas. You can define animations of the sprite sheet and then define the event to play the animations on click.

```
  <!-- defining media of a spritesheet -->
    <media id="walkAnim" src="animations/walkAnimation.png" type="spritesheet">
        <frames regX="0" regY="0" width="285" height="285" count="24" />
    <!-- Create the animations of the sprite sheet to play -->
        <animations>
            <walk>[0, 23, "stop",0.12]</walk>
            <stop>[23,23,"stop", 0.5]</stop>
        </animations>
    </media>

    <sprite asset="walkAnim" x="0" y="0" start="walk" id="walkAnimSprite" framerate="24">
    <!-- Create a event to play animation on click -->
        <event type="click">
            <action type="command" command="play" animation="walk" asset="walkAnimSprite" />
        </event>
    </sprite>
```

Links to external pages

Actions can be defined to open a external webpages from a content. This can be added to the ecml as a command of type "external" and href attribute containing the url of the web page to be opened.

The below ecml snippet opens the url "http://www.ekstep.org/" when clicked on the defined shape area.

```
   <shape type="rect" x="47" y="74" w="17" h="7" hitArea="true">
        <event type="click">
            <action type="command" command="external" href="http://www.ekstep.org/"/>
        </event>
    </shape>
```

Audio recording and processing using Sensibol

Recording audio of the child and processing using Sensibol or playback last recorded audio involves three steps. We are using sensibol API or Android media recorder to record audio. Using sensibol API we compares the recorded audio with the master audio and return a JSON result.

Sensibol uses a metadata file which is generated using master audio of the story/lesson. The metadata file is used while processing children audio. Each stage heading sentence/line is defined with index starting from 0. These index numbers are used while processing each stage recorded audio.

- `timeout` - To control recording time. default value is 10000 ms
- `timeout-success`, `timeout-failure` - To handle recording stop action on timeout. These values should be same as `stopRecord` action's `success`, `failure` values

Example:

- Today is the wedding of the elephant and the bear. Their jungle friends are celebrating. - 0
- The bear wore a saree. The elephant wore pants. - 1
- The eagle played the drums. The fox played the trumpet. - 2

We have created three commands to record and process child audio.

- startRecord - to start recording audio of the child.

```
   <action type="command" command="startRecord" asset="startrec" success="rec_started"
failure="rec_start_fail" timeout="1500" timeout-success="rec_stopped"
timeout-failure="rec_stop_failed"/>
    <!--
```

```
    asset - "id" of the start record container.
    success - event to dispatch on successful start of audio recording.
    failure - event to dispatch on failure of start audio recording.
    timeout - To control recording time. default value is 10000 ms.
    timeout-success, timeout-failure - To handle recording stop action on timeout. These values should be same as stopRecord action's success, failure values

    -->
```

- stopRecord - to stop recording.

```
   <action type="command" command="stopRecord" asset="stoprec" success="rec_stopped" failure="rec_stop_failed"/>
    <!--
    asset - "id" of the recording state container.
    success - event to dispatch on successful stop of audio recording.
    failure - event to dispatch on failure of stop audio recording.
    -->
```

- processRecord - to process recorded audio file

```
   <action type="command" command="processRecord" asset="stoprec" success="rec_processed" failure="rec_process_fail" data-lineindex="0"/>
    <!--
    asset - "id" of the recording state container.
    success - event to dispatch on successful processing of recorded audio.
    failure - event to dispatch on failure of processing recorded audio.
        data-lineindex - current recording audio line index from metadata file.
    -->
```

Below is the sample ecml of audio recording container:

```
<g id="recorder" x="0" y="0" w="100" h="100" visible="false">
    <g x="0" y="0" w="100" h="100">
        <image asset="blacktint" x="0" y="0" w="100" h="100">
            <event type="click">
                <action type="command" command="hide" asset="recorder" />
            </event>
        </image>
```

```
    </g>
    <g x="0" y="0" w="100" h="100" id="startrec">
        <shape type="roundrect" x="36" y="37" w="30" h="20" fill="gray"/>
        <shape type="roundrect" x="35" y="35" w="30" h="20" fill="#34e941">
            <event type="click">
                <action type="command" command="startRecord" timeout="1500" asset="startrec"
success="rec_started" failure="rec_start_fail"/>
            </event>
        </shape>
        <image asset="record" x="38" y="40" w="6" h="10"/>
        <text x="47" y="43" w="20" h="5" font="sans-serif" fontsize="50"
color="darkgreen">Start</text>
    </g>
    <g x="0" y="0" w="100" h="100" id="stoprec" visible="false">
        <shape type="roundrect" x="36" y="37" w="30" h="20" fill="gray"/>
        <shape type="roundrect" x="35" y="35" w="30" h="20" fill="#e93441">
            <event type="click">
                <action type="command" command="stopRecord" asset="stoprec"
success="rec_stopped" failure="rec_stop_failed"/>
            </event>
        </shape>
        <shape id="mover" type="circle" x="47" y="45" w="2" h="2" fill="#e98888"
visible="false"/>
        <image asset="stop_record" x="38" y="40" w="6" h="10"/>
    </g>
    <g x="0" y="0" w="100" h="100" id="welldone" visible="false">
        <shape type="roundrect" x="36" y="37" w="30" h="20" fill="gray"/>
        <shape type="roundrect" x="35" y="35" w="30" h="20" fill="#34e941">
            <event type="click">
                <action type="command" command="hide" asset="recorder" />
                <action type="command" command="hide" asset="welldone" />
                <action type="command" command="show" asset="startrec" />
            </event>
        </shape>
        <text x="47" y="43" w="20" h="5" font="sans-serif" fontsize="50" color="yellow">Well
Done</text>
    </g>
```

```
    <g x="0" y="0" w="100" h="100" id="tryagain" visible="false">
        <shape type="roundrect" x="36" y="37" w="30" h="20" fill="gray"/>
        <shape type="roundrect" x="35" y="35" w="30" h="20" fill="#34e941">
            <event type="click">
                <action type="command" command="startRecord" asset="tryagain"
success="rec_started" failure="rec_start_fail"/>
            </event>
        </shape>
        <image asset="record" x="38" y="40" w="6" h="10"/>
        <text x="47" y="43" w="20" h="5" font="sans-serif" fontsize="50" color="darkred">Try
Again</text>
    </g>
</g>

<appEvents
list="rec_started,rec_start_fail,rec_stopped,rec_stop_failed,rec_processed,rec_process_fail"/>

<events>
    <event type="rec_started">
        <action type="command" command="hide" asset="startrec" />
        <action type="command" command="hide" asset="tryagain" />
        <action type="command" command="show" asset="stoprec" />
        <action type="command" command="show" asset="mover" />
        <action type="animation" asset="mover">
            <tween id="mover" loop="true">
                <to ease="linear" duration="0">
                    <![CDATA[{"x":47,"y":45}]]>
                </to>
                <to ease="sineInOut" duration="1000">
                    <![CDATA[{"x":62,"y":45}]]>
                </to>
                <to ease="sineInOut" duration="1000">
                    <![CDATA[{"x":47,"y":45}]]>
                </to>
            </tween>
        </action>
    </event>
```

```
    <event type="rec_stopped">
        <action type="command" command="hide" asset="stoprec" />
        <action type="command" command="hide" asset="mover" />
        <action type="command" command="processRecord" asset="stoprec" success="rec_processed"
failure="rec_process_fail" data-lineindex="${stage.lineindex}"/>
    </event>
    <event type="rec_processed">
        <action type="command" command="show" asset="welldone" />
    </event>
    <event type="rec_process_fail">
        <action type="command" command="show" asset="tryagain" />
    </event>
</events>
```

Audio recording and playback last recorded audio

Always, the last recorded audio file is preloaded to memory by using `"current_rec"`identifier. we can use this as an asset in an action to play the audio. Below is the sample ECML for recording audio and playback.

```
   <g h="100" id="recorder" visible="false" w="100" x="0" y="0">
        <g h="100" w="100" x="0" y="0">
            <image asset="blacktint" h="100" w="100" x="0" y="0">
                <event type="click">
                    <action asset="recorder" command="hide" type="command"/>
                </event>
            </image>
        </g>
        <g h="100" id="startrec" w="100" x="0" y="0">
            <shape fill="gray" h="20" type="roundrect" w="30" x="36" y="37"/>
            <shape fill="#34e941" h="20" type="roundrect" w="30" x="35" y="35">
                <event type="click">
                    <action asset="startrec" command="startRecord" failure="rec_start_fail"
success="rec_started" type="command"/>
                </event>
            </shape>
```

```
            <image asset="record" h="10" w="6" x="38" y="40"/>
            <text color="darkgreen" font="sans-serif" fontsize="50" h="5" w="20" x="47"
y="43">Start</text>
        </g>
        <g h="100" id="stoprec" visible="false" w="100" x="0" y="0">
            <shape fill="gray" h="20" type="roundrect" w="30" x="36" y="37"/>
            <shape fill="#e93441" h="20" type="roundrect" w="30" x="35" y="35">
                <event type="click">
                    <action asset="stoprec" command="stopRecord" failure="rec_stop_failed"
success="rec_stopped" type="command"/>
                </event>
            </shape>
            <shape fill="#e98888" h="2" id="mover" type="circle" visible="false" w="2" x="47"
y="45"/>
            <image asset="stop_record" h="10" w="6" x="38" y="40"/>
        </g>
        <g h="100" id="playback" visible="false" w="100" x="0" y="0">
            <shape fill="gray" h="20" type="roundrect" w="30" x="36" y="37"/>
            <shape fill="#34e941" h="20" type="roundrect" w="30" x="35" y="35">
                <event type="click">
                    <action asset="current_rec" command="TOGGLEPLAY" type="command"/>
                    <action asset="recorder" command="hide" type="command"/>
                    <action asset="playback" command="hide" type="command"/>
                    <action asset="startrec" command="show" type="command"/>
                </event>
            </shape>
            <image asset="icon_sound" h="10" w="6" x="38" y="40"/>
            <text color="darkgreen" font="sans-serif" fontsize="50" h="5" w="20" x="47"
y="43">Play</text>
        </g>
    </g>
    <appEvents list="rec_started,rec_start_fail,rec_stopped,rec_stop_failed"/>
    <events>
        <event type="rec_started">
            <action asset="startrec" command="hide" type="command"/>
            <action asset="stoprec" command="show" type="command"/>
            <action asset="mover" command="show" type="command"/>
```

```
            <action asset="mover" type="animation">
                <tween id="mover" loop="true">
                    <to duration="0"
ease="linear"><![CDATA[{&quot;x&quot;:47,&quot;y&quot;:45}]]></to>
                    <to duration="1000"
ease="sineInOut"><![CDATA[{&quot;x&quot;:62,&quot;y&quot;:45}]]></to>
                    <to duration="1000"
ease="sineInOut"><![CDATA[{&quot;x&quot;:47,&quot;y&quot;:45}]]></to>
                </tween>
            </action>
        </event>
        <event type="rec_stopped">
            <action asset="stoprec" command="hide" type="command"/>
            <action asset="mover" command="hide" type="command"/>
            <action asset="playback" command="show" type="command"/>
        </event>
    </events>
```

Using Images

Image element uses media of type ‘image’ defined in manifest and it should use only relative path. Below are the list of attributes we can use with image element.

`id` - identifier of the element.

`asset` - identifier of the media of type ‘image’ defined in manifest.

`x`, `y` - location attributes.

`w`, `h` - bound attributes.

`visible` - to show/hide the elements on canvas. Boolean value. Default value is true.

The properties are relative to the parent container and specified in % to allow renderer to scale the scene with the device orientation and screen size.

Below is the sample usage of image element:

```
   <image asset="next" x="93" y="3" w="5" h="8.3" id="next" visible="false"/>
```

Using Text

Text element is used to render text on the canvas. It is part of a stage element. Below are the list of attributes we can use with text element.

`id` - identifier of the element.

`x`, `y` - location attributes.

`w`, `h` - bound attributes.

`font` - font family of the given text.

`fontsize` - size of the text font.

`align` - to align text. values are ‘left’, ‘right’ and ‘centre’. Default value is ‘left’.

`valign` - to align vertically. Values are ‘top’, ‘bottom’ and ‘middle’. Default value is 'top’.

`color` - color code to apply to the text. Default value is "#000000’.

`shadow` - color code to apply to the text shadow.

`blur` - to provide blur effect in text. works with shadow attribute.

`offsetX` - set X position offset in shadow of text, works with shadow attribute.

`offsetY` - set Y position offset in shadow of text, works with shadow attribute.

`weight` - set weight of the text. Values can be : italic, bold etc.

`visible` - to show/hide the elements on canvas. Boolean value. Default value is true.

`textBaseline` - allowed keywords/text correspond to alignment points(x,y) of the font/text. Default value is 'hanging'.

ref: https://html.spec.whatwg.org/multipage/scripting.html#dom-context-2d-textbaseline

There are three ways we can pass the text value to the element.

Direct:

```
   <text x="70.5" y="91" w="10" h="5" font="Arial" fontsize="28">Auto transition</text>
<text blur="10" color="white" fontsize="40" h="10" offsetY="5" outline="2" shadow="red" w="20" x="73" y="80">Text with Outline and Shadow</text>
    <text align="center" color="black" fontsize="100"  weight="italic" offsetY="10"  h="30" w="90" x="4" y="20">Text Plugin</text>
```

Using param:

```
   <param name="body" model="storyData.scene1_body"/>
    <text param="body" x="9" y="7" w="86" h="4" font="Georgia" fontsize="42"/>
```

Using model:

```
   <text model="storyData.kettle_title" x="8" y="17" w="86" h="4" font="Georgia" weight="bold" fontsize="128"/>
```

Note: For Indian languages, specific fonts need to be used to ensure that the text renders correctly on Android devices. Please see the ContentApp Supported Fonts.

Using Shape

Shape element is used to draw different type of shapes on canvas. It is part of a stage element. Below are the list of attributes we can use with shape element.

`id` - identifier of the element.

`type` - shape type value. Values are ‘rect’, ‘roundrect’ and ‘circle’.

`x`, `y` - location attributes.

`w`, `h` - bound attributes.

`opacity` - to set the opacity level of the element.

`visible` - to show/hide the elements on canvas. Boolean value. Default value is true.

`fill` - the color code to fill.

`stroke` - the color code to stroke.

`radius` - radius for rounded rectangle.

`stroke-width` - thickness of the stroke.

Below is the sample usage of shape element:

```
  <shape x="25" y="50" h="10" w="15" type="rect" stroke="white"/>
```

Using Scribble

Scribble element is used to create a scribble pad on canvas. It is part of a stage element. Below are the list of attributes we can use with scribble element.

`id` - identifier of the element.

`x`, `y` - location attributes.

`w`, `h` - bound attributes.

`opacity` - to set the opacity level of the element.

`color` - to set the color of paint brush.

`fill` - the color code to fill.

`stroke` - the color code to stroke.

`thickness` - to set thickness of paint brush.

`stroke-width` - thickness of the stroke.

Below is the sample usage of shape element:

```
   <scribble id="scribblePad" stroke="#ABABAB" stroke-width="3" z-index="1" h="46.5" w="86"
x="7.1" y="48" opacity="1" fill="black" color="white" thickness="5" />
```

To erase the scribble pad area: you can bind event with any display object.

```
   <event type="click">
        <action type="command" command="erase" asset="scribblePad" /> // asset should be id of
the scribble to erase.
    </event>
```

Delay an action

We can pause an action registered with an event for a specific time by using `delay`attribute.

Below is the sample ECML to show an image after 10 sec of entering to the stage.

```
   <events>
        <event type="enter">
                <action type="command" command="toggleShow" asset="scene1_body" delay="10000"
/>
        </event>
    </events>
```

Using HTML

To render html over the canvas, a new tag `<div>` was introduced. It supports "id", "x", "y", "w", "h" and "style" attributes.

`id` - identifier of the element.

`x`, `y` - location attributes.

`w`, `h` - bound attributes.

`style` - supports all valid css styling.

Sample usages:

Use CDATA for html code.

```
   <div id="testDiv" x="10" y="30" w="40" h="5" style="color:red;
background:green;text-align:center;">
        <![CDATA[<div><span style="font-weight:bold;">This </span><span>is </span><span>the
</span><span id="annualHeader">HTML </span><span>div </span><span>Element</span></div>]]>
    </div>
```

You also can bind event with tag just like in other tag. See the example below:

```
   <div id="testDiv" x="10" y="30" w="40" h="5" style="color:red;
background:green;text-align:center;">
        <![CDATA[<div><span style="font-weight:bold;">This </span><span>is </span><span>the
</span><span id="annualHeader">HTML </span><span>div </span><span>Element</span></div>]]>
        <event type="click">
            <action type="command" command="toggleshow" asset="testShape"/>
        </event>
    </div>
```

Hotspot on HTML content

You can use anchor tag with `data-event` attribute to make any text as hotspot inside HTML content. Below is the sample for the same.

```
   <image asset="page1" x="0" y="0" w="100" h="100" />

    <div id="testDiv" x="10" y="30" w="40" h="50" style="color:red;
background:green;text-align:center;">
        <![CDATA[ <a href="javascript:void(0)" data-event="show_bg_image">Show BG</a> <a
href="javascript:void(0)" data-event="hide_bg_image">Hide BG</a> ]]>
```

```
    </div>
    <appEvents list="show_bg_image,hide_bg_image"/>
    <events>
        <event type="show_bg_image">
            <action asset="page1" command="show" type="command"/>
        </event>
         <event type="hide_bg_image">
            <action asset="page1" command="hide" type="command"/>
        </event>
    </events>
```

Using Video

We are using 'video' tag to play video on canvas. We are creating HTML5 'video' element on top of canvas(Similar to 'div' element).

Currently we are supporting 2 video formats.

1. .ogv - formatType 'video/ogg'.

2. .webm - formatType 'video/webm'.

Below are the list of attributes we can use with video element. `id` - identifier of the element.

`x`, `y` - location attributes.

`w`, `h` - bound attributes.

`type` - to specify video formattype (webm/ogg).

`asset` - identifier of the media of type ‘video’ defined in manifest.

`autoplay` - (true) auto play of video after rendering.

`controls` - (true) to show default video controls.

Event Commands:

`play` - to play video specified in `asset`(media id specfied as type `video`).

`pause` - to pause video specified in `asset`(media id specfied as type `video`).

```
 <manifest>
    <media id="rainGoAway" src="video/rain_go_away.ogv" type="video" />
  </manifest>

  <stage>
    <video asset="rainGoAway" type="video/ogg" x="10" y="22" w="40" h="30" autoplay="true"
controls="false"/>
    <event type="click">
         <action type="command" command="play" asset="rainGoAway"/>
     </event>

  </stage>
```

Using Grid

We are using 'grid' tag to provide layout.

Below are the list of attributes we can use with video element.

`id` - identifier of the element.

`x`, `y` - location attributes.

`w`, `h` - bound attributes.

`cols` - number of column.

`iterate` - iterate through the json list.

```
 <grid cols="2" h="20" id="grid1" iterate="testdata3.users" var="user" w="30" x="32" y="47">
    <shape fill="#0099FF " h="100" stroke="black" type="rect" w="100" x="0" y="0"/>
    <text color="black" fontsize="600" h="90" model="user.name" valign="middle" w="90" x="40"
```

```
y="10"/>
  </grid>
```

Show and Hide HTML Elements

We can show and hide the html elements of a stage with the below commands. We use these commands when the canvas layer elements like popups are hidden by html elements.

- SHOWHTMLELEMENTS - To show html elements of the stage.
- HIDEHTMLELEMENTS - To hide html elements of the stage.

Below is the sample ECML:

```
   <stage extends="baseStage" h="100" id="hybridRendering" w="100" x="0" y="0">
        <param name="next" value="allFontRendering"/>
        <param name="previous" value="scribbleTest"/>
        <div id="testDiv" x="10" y="20" w="80" h="35" style="color:white; text-align:center;
font-family: NotoSansTelugu, arial; font-size:35px;">
            <![CDATA[<span style="font-weight:bold;">This is a HTML element.</span>]]>
        </div>
        <shape id="circleShape" x="35" y="70" w="10" h="10" type="circle" fill="green">
            <event type="click">
                <action type="command" command="showhtmlelements" asset="hybridRendering"/>
            </event>
        </shape>
        <shape id="circleShape" x="65" y="70" w="10" h="10" type="circle" fill="red">
            <event type="click">
                <action type="command" command="hidehtmlelements" asset="hybridRendering"/>
            </event>
        </shape>
    </stage>
```

Teacher Instructions

This helps to show "Teacher instructions" on each stage wise. Teacher instructions information for the stage is a part of 'stage' ECML/JSON data. We have to use "param" tag with the name "instructions" inside stage.

Below is the sample ECML:

```
<stage id="scene1" x="0" y="0" w="100" h="100">
  <param name="previous" value="splash"/>
  <param name="next" value="scene2"/>
  <param name="instructions" value="Scene 1 instructions"/>
  ...
</stage>
```