# video player (how to get ahold of a component in shadow DOM)

Given a component without any imperative DOM manipulation, how would one use a video tag to play a video with source and volume controlled via components model.

|  |
| --- |
| <template>  <ul>  <li ng-repeat="video in videos" on-click="selectVideo(video)"></li>  </ul>  <video ng-if **id="player"**></video>  </template>  class VideoGalleryController {  constructor(videos, @inject("player") player) {  this.videos = videos;  this.player = player;  }  selectVideo(video){  this.player.src = video.url;  this.**player**.play();  }  } |

# tabs (how to get ahold of child components in light DOM)

|  |
| --- |
| **Usage:**  <tab-container>  <tab-pane title="Tab1">Hello Tab1</tab-pane>  <tab-pane ng-repeat title="Tab2">Hello Tab2</tab-pane>  </tab-container>  **TabContainer:**  <template>  <div ng-repeat="tab in tabs" ng-click="select(tab)">{{tab.title}}</div>  <content/>  </template>  @QueryChildren(  'tabs': {type: TabPane, selector: 'tab-pane'}  ])  class TabContainer {  selectTab(tab) {  if (this.selectedTab) { this.selectedTab.selected = false; }  this.selectedTab = tab;  tab.selected = true;  }  }  **TabPane:**  <template>  <content ng-if="selected"/>  </template>  @Bind({  'title': 'title'  })  class TabPane {    } |

# ng-model-options (getting ahold of parent component)

|  |
| --- |
| **Usage:**  <form ng-model-options="{updateOn: 'blur'}">  <input ng-model="someModel">  </form>  @QueryParent(  'ngModelOptions': {type: NgModelOptions, selector: 'ng-model-options'}  ])  class NgModel {  @observe('ngModelOptions')  ngModelOptionsChanged(newVal) {  this.element.removeEventListener(this.eventName, this.updateValue));  this.eventName = newVal.updateOn;  this.element.addEventListener(this.eventName, this.updateValue);  }  } |

# validators (getting ahold of component or directive instances on a the same element)

|  |
| --- |
| Usage:  <input bind-ng-model="validatedValue" required min-length="2">  Validated value: {{ validatedValue }}  @QueryThis(  'validators': {type: Validator, collection: true}  ])  @Bind({  'value': 'value'  })  class NgModel {  valueChanged(value) {  var valid =  this.validators.reduce((valid, validator) =>   valid = valid && validator.validate(value),  true);  this.ngModel = valid ? value : null;  }  }  class Validator {  validate(value) => { return true; }  }  class MinLengthValidator extends Validator {  validate(value) {  return value.length < 10;  }  } |

# Transitive import and third party custom elements

Create a custom element that uses an XTags toggle button internally and make the usage of the custom element easy.

|  |
| --- |
| **Usage:**  <link rel="import" href="todo-list.html">  <todo-list items="[{title:'milk'}, {title: 'eggs'}]"></todo-list>  **todo-list.html:**  <link rel="import" href="xtags.html"></link>  <angular-element name="todo-list">  <template>  <template ng-repeat="item in items">  <x-toggle bind-checked="item.done">{{item.title}}</x-toggle>  </template>  </template>  </angular-element>  **x-tags.html:**  <script src="xtags.js"></script>  <link rel="stylesheet" href="xtags.css"></link> |

# assemble tab container with a Fwk other than Angular

|  |
| --- |
| **Usage in Vanilla JS:**  var tc = document.createElement('tab-container');  tc.innerHTML = '<tab-pane title="Tab1">Hello Tab1</tab-pane>  <tab-pane title="Tab2">Hello Tab2</tab-pane>';  document.body.appendChild(tc);  var tabPane = document.createElement('tab-pane');  tabPane.selected = true;  tc.appendChild(tabPane);  **Usage in Polymer:**  <tab-container>  <template repeat="l in list">  <tab-pane …>  </template>  </tab-container>  **TabContainer:**  <template>  <div ng-repeat="tab in tabs" ng-click="select(tab)">{{tab.title}}</div>  <content/>  </template>  class TabContainer { … } // see above  class TabPanel { … } // see above |

# wrap jquery plugin

Create a component that provides a jquery ui button as a custom element.

See <https://jqueryui.com/button/>

# ng-show

Modify the style of a dom node directly via a decorator directive

-> how should directives/components access the underlying node?

-> use decorator directives (and template directives) only in Angular templates, but not outside!

|  |
| --- |
| <template>  <button ng-click="shown = !shown">Toggle show</button>  <div ng-show="shown">Shown</div>  </template> |

# routing

Given two components create a routing component that is able to swap them.

|  |
| --- |
| <template>  <button ng-click="route.goToRoute1()>Set route 1</button>  <button ng-click="route.goToRoute2()">Set route 2</button>  <ng-view name="foo"></div>  </template>  Route1:  <template>  Route1  </template>  Route2:  <template>  Route2  </template> |

# dialogs

Show and hide dialogs as overlays to the current application. The dialogs should be defined as top level components (e.g. directly below <body>), so they are not a parent element of the element that wants to show them.

|  |
| --- |
| ?? Implement via routing ?? |

# 3rd party components with isolated state and configuration

3rd party component developer building reusable components should be able to configure services in a way that will be preserved even if the same services are configured differently by the host application. Examples of this scenario would be custom authentication and configuration of

# 3rd party components with overriden state and configuration

Application developer using 3rd party components might want to be able to reach into the components and reconfigure some of the services to work in the host environment. Examples of this would be custom authentication, encryption of local storage, etc.

# ~~Interacting with injectors of parent components~~

We need to be able to have two or more instances of the same component configured differently and how these different instances share or not share configuration.

Should the component have its own injector which interacts with the main injector of the application or should it interact with injector of the nearest parent component?

Problems with the parent injector approach:

- Calling document.createElement we can't inject the dependencies because we don't know what the parent is. That means that even if you create the element, you can't interact with it until it is somehow injected (possibly by adding it to the DOM).

- Since parent's might change during the life-time of a component, it's not clear how this would affect the injected dependencies, which are usually injected only once in the constructor.

<my-app>

#shadow

<github-config user="{{ user }}">

<github-issue-list issues="{{issues}}">

#shadow

<plus-one-button>

<my-app>

<my-list>

#shadow

<tab-container>

<tab-pane>

<tab-pane>

<tab-pane>

<tab-container>

<tab-pane>

<tab-pane>

<tab-pane>

<your-list>

#shadow

<tab-container>

<tab-pane>

<tab-pane>

<tab-pane>

# Scalability: Simple app should be simple, complex app should be manageable

Hello world should be simple

# Creating primitive components with templatable content

Component developer is creating general use components that might require expected nesting and sharing of templates between the child components and parent component.  
  
This is generally simple when we’re talking about something like a tab component, where the child components are responsible for rendering their own templated (transcluded) content.

This becomes more difficult when the parent component is responsible for rendering the child component’s content against a context or scope that is determined by the parent. Examples of this occur in situations as follows:  
  
The templated product order table (form/input example added to illustrate possible complexity):  
  
<custom-table row-source=”product in products”>

<custom-table-cell headertext=”Product”>{{product.name}}</custom-table-cell>

<custom-table-cell headertext=”Price”>{{product.price | currency}} / ea.</custom-table-cell>

<custom-table-cell headertext=”Quantity” ng-form=”quantity”>

<input type=”number” min=”0” ng-model=”product.quantity” name=”quantity”/>

</custom-table-cell>

</custom-table>  
  
Where the expected HTML result would look something like:  
  
<table>  
 <thead>

<tr>

<th>Product</th>

<th>Price</th>

<th>Quantity</th>

</tr>

</thead>

<tbody>

<tr>

<td>12-pack Beverages</td>

<td>$3.89 / ea.</td>

<td>

<input type=”number” min=”0” name=”quantity” value=”0”/>

</td>

</tr>

<tr>

<td>Bag o’ Chips</td>

<td>$1.39 / ea.</td>

<td>

<input type=”number” min=”0” name=”quantity” value=”0”/>

</td>

</tr>

</tbody>

</table>

Another example of these primitive types could be creating markup for controlling canvas items. For example the following might create a scene with WebGL, and render it to a canvas with a light source, a sphere and cylinder and some templated text, all from a specified camera angle:  
  
<three-d-scene>

<three-d-spotlight x=”0” y=”0” z=”400”/>

<three-d-sphere x=”0” y=”50” z=”150” r=”50” color=”red”/>

<three-d-cylinder x=”10” y=”23” z=”123” r=”25” color=”green”/>

<three-d-text>This is {{some}} text</three-d-text>

<three-d-camera x=”100” y=”300” z=”0” pointAtX=”0” pointAtY=”0”/>

</three-d-scene>  
  
This will be difficult because A) the children don’t map at all to actual DOM elements, rather constructs in JavaScript that is controlling a canvas. and B) The template from the <three-d-text> node will need to be shared and evaluated by the parent, which relates to an actual <canvas> element.