Angular 2.0: Life of a Template

Hello Anonymous! How are you? Just want to let you know that we are doing a really rough draft of this document right now. So, it might not be coherent or even correct until we are done :) Just letting you know...

# API

When a developer creates a component...

1. Developer authors the component code
2. Pass component Type with Metadata to compiler
3. Compiler returns ProtoView (via Promise)
4. Create ViewFactory from ProtoView (maybe)
5. Use ProtoView/ViewFactory to instantiate Views

*Note: Steps 2 - 5 usually happen transparent to the developer.*

# Compiler Internals

When the compiler is invoked to compile a component...

1. Extract directive/formatter configuration from the component metadata.
   1. From the list of directives build Selector engine
   2. recursively compile any components and their templates
      1. Note, this may actually happen later, as we don't want to fetch templates for components which are imported but not used.
2. Recursively walk the elements
   1. If element has directive/binding/events/etc...
      1. Mark elements with ng-binding class
      2. Create ProtoElementInjector and configure it with metadata
3. Return a ProtoView (via Promise)

# Component Instantiation

When a component instance needs to be created...

1. Clone Template
2. getElementsByClassName('ng-binding') for Elements with Marker Class
3. Match up elements to ProtoElementInjector and create ElementInjector for each element which contains binding.
4. Return View

## 

## Definition

Let's start with a component and its template.

|  |
| --- |
| import \* as core from 'angular/core';  class Item {  text:string;  constructor(text) {  this.text = text;  }  }  @Component({  selector: 'todos',  template: new TemplateConfig({  url: 'todos.html',  uses: [core.directives]  })  })  class Todos {  todos:List<Item>;  constructor() {  this.todos = [new Item('Take over the World')];  }  } |

|  |
| --- |
| <ul>  <li [ng-repeat|item]="todos">  ${item.text}  </li>  </ul>  <span>There are ${todos.length} todos.</span> |

## Understanding the compilation API

|  |
| --- |
| var type:Type = Todos;  var Promise<ProtoView> protoViewPromise = compile(type); |

Notice that the compilation is on the component rather than on the DOM element. This is because the DOM element is not sufficient by itself to define the UI. The key missing parts is the configuration of which directives are active in the current template. This is information is present in the Component annotation.

The second thing to notice, is that the result of compilation is a promise rather than a ready to use ProtoView. This is because components may load other components and loading is async. This is in contrast to current implementation where the compilation is synchronous but the runtime (instantiation of components) is asynchronous.

## Configuring the Compiler

The compiler's job is to look for directives in the template and to create ProtoView. The ProtoView is a serializable description of the template, its configuration and its directives. To achieve the compilation the compiler needs to configure which directives should be active for a given component. This information is defined in the components @Component annotation.

## Compilation Process

|  |
| --- |
| <ul>  <li [ng-repeat|item]="todos">  ${item.text}  </li>  </ul>  <span>There are ${todos.length} todos.</span> |

### NgRepeat: ProtoView Template

|  |
| --- |
| <li class="ng-bindable" [text|0]="item.text"></li> |
| new ProtoView({  type: LocalContext,  protoElementInjectors: [  new ProtoElementInjector({  textNodes: [ 0 ]  })  ],  protoWatchGroup: new ProtoWatchGroup({  'item.text': 0  })  }); |
| new View(  documentFragment: ...  elementInjectors: [],  textNodes: [ #text ]  ) |

### Component: ProtoView Template

|  |
| --- |
| <ul>  <template class="ng-binding"></template>  </ul>  <span class="ng-binding"  [text|0]="'There are ' + todos.length + 'todos.'"></span> |
| new ProtoView({  type: Todos,  protoElementInjectors: [  new ProtoElementInjector({  directives: [ NgRepeat, liProtoView ]  }),  new ProtoElementInjector({  textNodes: [ 0 ]  })  ],  protoWatchGroup: new ProtoWatchGroup({  'todos': new DirectiveToken(0, 0, setterFn),  // setterFn(o, v) => o.collection = v;  "'There are ' + todos.length + 'todos.'": 0  })  }) |
| new View({  documentFragment: ...  elementInjectors: [  new ElementInjector( <template> )  ],  textNodes: [ #text ]  }) |

## Walker

|  |
| --- |
| class Walker {  selector: Selector;  protoElementInjectors:Array<ProtoElementInjectors>;  protoWatchGroup:ProtoWatchGroup  textNodeIndex:int;  visit(element:Element) {  // We processe the child text nodes and hoist their bindings  var childElements:Array<Element> = [];  var protoElementInjector = null;  element.forEach((node, index) => {  switch (node.nodeType) {  case Document.ELEMENT\_TYPE:  childElements.push(node);  break;  case Document.TEXT\_TYPE;  var interpolation = interpolate(node.text);  if (interpolation) {  protoElementInjector =  ensureProtoElementInjector(protoElementInjector);  protoElementInjector.textNode.push(index);  protoWatchGroup.watch(  interpolation, this.textNodeIndex++);  }  break;  }  });  // Look for bindings on the current Node  var directives:Array<TypeDirectiveTuple>;  directives = selector.select(element);  if (directives[0].isTemplate) {  protoElementInjector.addDirective(directives.shift());  var walker = new Walker(element);  walker.setupProtoElementInjector(childElements, directives);  protoElementInjector.protoView = walker.buildProtoView();  } else {  setupProtoElementInjector(childElements, directives);  }  }  setupProtoElementInjector(childElements:Array<Element>,  directives:Array<TypeDirectiveTuple>)  {  directives.forEach(tuple => {  if (tuple.annotation is Component) {  // recurse into the components template  }  protoElementInjector.addDirective(tuple.type);  });  childElements.forEach(visit);  }  ensureProtoElementInjector(protoElementInjector) {  if (protoElementInjector == null) {  protoElementInjector = new ProtoElementInjector();  this.protoElementInjectors.push(proto);  }  return protoElementInjector;  }  } |

|  |
| --- |
| @Template({  selector: '[ng-repeat]',  //observe: {'ng-repeat': 'onCollectionChange'}  })  class NgRepeat {  constructor(viewFactor:ViewFactor, viewPort:ViewPort) {  }  @Observe('ng-repeat')  onCollectionChange(record:ChangeRecord) {  record.forEachNew((e) => {  var view:View = viewFactory.new();  viewPort.add(view);  });  record.forEachDelete((e) => {  view.destroy();  });  }  } |

|  |
| --- |
| <app>  #ShadowRoot  <tab>  #ShadowRoot  \*  <pane>  #ShadowRoot  \*  </pane>  <tab> |