Design: Component Interactions in Angular

*Status:* ***(Draft)***

*Authors: misko@google.com*

*This document is published to the web as part of the public* [*Angular Design Docs*](https://drive.google.com/#folders/0BxgtL8yFJbacUnUxc3l5aTZrbVk) *folder*

# Objective

Currently, components interact with each other via Scope.on and Scope.broadcast. Since event handling is being decoupled from Scope, we need other mechanisms to facilitate these interactions.

This document describes the ways components interact with each other and how to implement these interactions without Scope events.

# Background

When component controller becomes the context, than component has no good way of getting hold of Scope. This is because Scope needs the controller for the context. This circular dependency needs to be broken.

A big reason why components inject Scope is that they need to do event handling. So removing event handling from Scope would break the circular dependency.

This requires us to rethink how components interact with each other, because right now many of them do it via Scope.on and Scope.broadcast.

### Goals

Identity clear ways to implement interactions between components and provide all the necessary tools to implement such interactions.

# Prior Art

AngularJS

Polymer [communications](http://www.polymer-project.org/articles/communication.html). Polymer uses DOM events to communicate up the DOM tree and querySelectorAll to communicate down.

# Detailed Design

Scope event handling have three separate responsibilities: on, emit, broadcast. These will be broken of from Scope and solved in different ways.

## Scope.on and Scope.emit

The `Scope.on` and `Scope.emit` methods are used to enable the communication of components in the application. The four main scenarios are

### When Component and SubComponent Form an Aggregate

SOLUTION: Use Dependency Injection

In this scenario, Component cannot be used without SubComponent. Similarly, SubComponent is used only in the context of Component. These two components are usually developed by the same group of people. And the client mostly interacts with Component.

Example:

|  |  |
| --- | --- |
| Light DOM:  <sortable-list header="Countries">  <item>USA</item>  <item>Canada</item>  </sortable-list> | Shadow DOM of SortableList:  <sortable-header>{{header}}</sortable-header>  <!-- render items -->> |

As you can see, SortableHeader is a part of SortableList's implementation. Since these two components have to know about each other, using events does not buy us anything.

Instead, we should use dependency injection.

|  |
| --- |
| class SortableList {  void sort(order){}  }  class SortableHeader {  final SortableList sortableList;  SortableHeader(this.sortableList);  } |

When SortableHeader wants to notify the list about a change, it does not fire an event -- it calls `sort` directly.

|  |
| --- |
| collection.sort("asc"); |

If you would like to reuse SortableHeader in some other component, you can create an interface that SortableList implements.

|  |
| --- |
| class SortableList implements SortableCollection {  void sort(order){}  }  class FancySortableList implements SortableCollection {  void sort(order){}  }  class SortableHeader {  final SortableCollection collection; // can be SortableList or FancySortableList  SortableHeader(this.collection);  } |

The benefits of this solution are

* The code is easy to follow.
* The solution is IDE-friendly.

**Implementation**

It should work already.

### When SubComponent is Used By The Author of Component

SOLUTION: Use [Events](#h.osc3pbxis5i0)

In this scenario Component is written by Author A, SubComponent is written by Author B. SubComponent is used in the template of Component. Since SubComponent is authored by a different person, it cannot inject an instance of Component. We have to use events.

Example:

|  |  |
| --- | --- |
| Light DOM:  <sortable-list header="Countries">  <item>USA</item>  <item>Canada</item>  </sortable-list> | Shadow DOM of SortableList:  <toggle (onChange)="toggleSorting()">{{header}}</toggle>  <!-- render items -->> |

The toggle component is generic. And the SortableList component looks as follows:

|  |
| --- |
| class SortableList {  void toggleSorting(){}  } |

**When Component and SubComponent Are Composed By a Third Author**

*[Note to readers - the solution described here is still provisional and under discussion.]*

SOLUTION: Access Component Using `#` Syntax

In this scenario Component is written by Author A, SubComponent is written by Author B, and the two components are composed by Author C.

Example:

|  |
| --- |
| <zippy title="Greeting">  Body of the text which is shown conditionally.  <blinking-button>Toggle</blinking-button>  </zippy> |

Zippy and BlinkingButton are two generic components authored by different people. They do not know about each other. Therefore, we cannot use dependency injection because it will require some agreement ahead of time.

Currently in Angular it is possible for a SubComponent to reference Component, when SubComponent is part of Component's shadow DOM. In this example, however, BlinkingButton, which is SubComponent, is part of Zippy's light DOM, and there is no mechanism that allows BlinkingButton to reference Zippy in the template. To overcome this limitation, we will provide a way to publish the controller of Zippy in the light DOM using the following syntax:

|  |
| --- |
| <zippy #zippy=fieldName title="Greeting">  Body of the text which is shown conditionally.  <blinking-button (click)="zippy.toggle()">Toggle</blinking-button>  </zippy> |

The #zippy is analogous to *id* in html but is scoped to the Angular view.

The benefits of this solution are

* No need to use adapters or communicate via the parent component.
* It is easy to see how the components are wired up.

### Implementation

* Change the compiler to add support for the `# syntax`.

**Component and SubComponent in LightDOM**

**Other Solutions**

Without introducing the new syntax, wiring up `zippy` and `blinking-button` will require using the encompassing component.

Example:

|  |
| --- |
| Shadow DOM of EncompassingComponent:  <zippy title="Greeting" [is-open]="is-open">  Body of the text which is shown conditionally.  <blinking-button (click)="isOpen = !isOpen">Toggle</blinking-button>  </zippy>  class EncompassingComponent {  @Property('is-open')  bool isOpen;  void toggleZippy() => !isOpen;  } |

This gets more complicated if you imagine, for instance, a collection of `zippy` rendered using an ng-repeat. On top of that, if zippy does not provide the `is-open` property, we will have to use the Query API to actually obtain an instance of `zippy`.

**When Component Emits Application Level Events**

SOLUTION: Use a Message Bus Service

In this scenario Component wants to send an application-level event. Let's say we have a component that submits a message to a social network. And when a new message is submitted, we want to update other components in the app (e.g., some counters). To do that the component should inject a message bus service and send an event using the bus. Other components interesting in this event should inject the bus and attach a listener.

* Such events are application-scoped and should not depend on where the component is located in the DOM. Thus, using DOM events is not acceptable.
* The message bus doesn’t need to be part of Angular. It could be 3rd party or application-specific.

## Scope.broadcast

*[Note to readers - the entire scope.broadcast section is still provisional and under discussion.]*

Scope broadcast is unique, since we believe that it is best if we no longer support broadcasting of events. When looking over common usages of the dispatch, we have found that it is often used when the parent needs to communicate with the children. A better way to do so would be for parent to have a list of children and then just be able to directly call them by invoking a specific API.

|  |
| --- |
| @Component(  selector: 'tabs',  template: '<ul><li [ng-repeat|p]="panes">${p.title}</li></ul>'  )  class Tabs {  @children Query<Pane> panes;    Tabs(this.panes) {}  onEvent() {  panes.forEach((p) => p.notify());  }  }  @Component(  selector: 'pane',  map: const {'title': 'title'}  )  class Pane {  String title;  notify() {  // doSomething  }  }  <tabs>  <pane title="Overview"></pane>  <pane [ng-repeat|p]="paneList" [title]="p.name"></pane>  <tabs> |

### Query API

The Query class should implement the Iterable interface

|  |
| --- |
| class Query<T> implements Iterable<T> {  void listen(listener) {}  } |

Query should also have the listen method, so Component can react to changes.

|  |
| --- |
| query.listen(() {  query.forEach(doStuff);  }); |

### Query Visibility

The Query API will use annotations to specify what needs to be traversed when a Query executes. The following are all the options that will be supported:

* @children - only direct children of the parent component in the light DOM.
* @descendants - all descendants of the parent component in the light DOM.
* @shadow @children - only direct children of the parent component in the shadow DOM.
* @shadow @descendants - all descendants of the parent component in the shadow DOM.

There is also an option called “self”. It cannot be specified by the developer, but will be used internally by the framework.

The query does not cross component boundaries. So the @children and @descendants annotations are used to get components from the light DOM. And @shadow is used to get components from the shadow DOM. There is no way to get components from the composed DOM.

**Implementation**

* Change DI to support multiple annotations.
* Improve DI’s support of parameterized types. At the moment you can only access the Query<Child> type, whereas we need Query and Child.
* Build and maintain trees of directive injectors. We cannot traverse the DOM to execute a Query. The closest thing we can do is to traverse the trees of directive injectors.
* Change DirectiveInjector to instantiate and provide Query objects when needed.
* Change DirectiveInjector to notify affected queries when a new component gets created.
* Change ViewPort to notify affected queries when a View gets moved or removed.
* Implement the Query class.

## Template Events

We can use the `()` syntax to register an event handler.

Example:

|  |
| --- |
| <tabs (change)="onContactSelection(event)">  <pane name="Overview" [value]="null">...</pane>  <pane [ng-repeat|contact]="contacts" [name]="contact.name" [value]="contact">  ${p.text}  </pane>  </tabs> |

If tabs is a web component, then the change event is a DOM event that is triggered on the tabs element. So when using `()` we do not handle bubbling events. The reason for that is to make the syntax work the same way for Web components and Angular components.

Now, suppose `tabs` is an Angular component. In this case, we should use the EventEmitter annotation to inject a function emitting the change event.

|  |
| --- |
| class Tabs {  Function emitChange;    Tabs(@EventEmitter('change') this.emitChange);  // Pane get selectedPane => ...;  }  // To emit an event: emitChange(selectedPane); |

We can also use the `(^)` syntax to register an event handler on an element.

Example:

|  |
| --- |
| <div class="tabs">  <div [ng-repeat|pane]="panes" class="tab" (^click)="select(pane)">  <img [src]="pane.icon"><span>${tab.name}</span>  </div>  </div> |

A event handler registered using `(^)` handles bubbling events. In the example above, the click handler will be called regardless of the target of the event, as long as it bubbles up to the div.

The benefits of this solution are

* We can use Angular components and Web components in a similar fashion in the template.
* When using Angular components, an event firing is just a method call, which is fast.

**Implementation**

* Change DI to support annotation instances, not just annotation types. Currently the DI library preserves only annotation types. This means that @EventEmitter('change') and @EventEmitter('click') are identical for the DI library, and the only thing the Angular can get is the EventEmitter type.