One Time / Labeled Bindings

*Status: (Draft)*

*Authors:* [*kkseamon@google.com*](mailto:kkseamon@google.com)*,* [*igor@angularjs.org*](mailto:igor@angularjs.org)*,* [*caitpotter88@gmail.com*](mailto:caitpotter88@gmail.com)

Objective

Improve performance of Angular apps by providing a new kind of binding that would be "frozen" after the first time the model value propagates through the binding.

Background

Data-binding makes application very dynamic and fluid by nature, but it also has some cost. When creating very complex views with lots of data-bound points it's common that after the initial rendering the view doesn't change. If the templating system was aware of this static nature of the view, it could use this information to perform performance optimizations and free up resources once the view has been initialized.

Prior Art

# Bind once

Simple example: [fast-bind-once](https://github.com/kseamon/fast-bind/blob/master/src/directives/bind-once/bind-once.js)

Sets text once at link time.

### Benefits

Fast - no watchers registered.

### Downsides/alternatives

Not usable for asynchronously populated content.

Ways to address this

* Add an optional “don’t bind until” attribute (We use this approach in croc’s stable bindings)
* $watch and update the value until it is defined (and maybe not null) (at the cost of performance in some cases)

### Other notes

fast-bind-once could be slightly faster if it was terminal and used textContent instead of jqLite.text(), but there may be trade-offs.

# Bind on notify

Simple example: [fast-bind-on-notify](https://github.com/kseamon/fast-bind/blob/master/src/directives/bind-on-notify/bind-on-notify.js) and [fast-bind-notifier](https://github.com/kseamon/fast-bind/blob/master/src/directives/bind-notifier/bind-notifier.js)

fast-bind-on-notify sets text whenever the expression watched by fast-bind-notifier updates

### Benefits

Allows many bindings to update in response to a single watcher. Useful in paginated or virtually scrolling repeaters in particular.

### Downsides/alternatives

As with bind-once, individual bindings may not work properly with asynchronously populated content. Alternative options that address this are the same.

### Other notes

croc’s stable bindings use an approach similar to this, with a couple differences that we probably do not want to replicate

* Instead of a notifier directive, there is a special repeater that replaces ng-repeat and sends notifications when its watched array updates
* It sends notifications by calling functions in an array on the scope rather than using angular’s $broadcast.

# $apply isolation

croc (an internal component library) has a service that allows creation of a scope S with a monkey-patched $apply. This version of $apply $digests $rootScope, S, and S’s descendents (To this end, it temporarily removes other scopes from the tree).

### Benefits

Allows for much faster $digest cycles in isolated areas of the page (dialogs, panes, etc) due to unrelated bindings not being checked.

### Downsides/alternatives

Adds a lot of new ways to “mess up”. Communication between this scope and others is awkward. An approach that was baked into Angular rather than hacked on could probably do better. Bind regions may be a cleaner alternative.

# AngularDart and one time bindings

AngularDart has two kinds of qualified bindings, a lazy and non-null. The watching mechanism [was extended](https://github.com/angular/angular.dart/blob/5ad9c7a0529ec9166948ab6833403bec4901dd66/lib/core/scope.dart#L226-L238) to directly support them via syntax:

* lazy: {{:: user.name }} - the binding is disabled as soon as binding evaluates to a non-null value
* non-null: {{: user.name }} - the reaction fn is called only when this binding evaluates to a non-null value - this has nothing to do with one-time binding so we don't need to consider it and it's mentioned here just for the sake of completeness.

Additionally there is a [binding mapping annotation](https://github.com/angular/angular.dart/blob/f7115aa86da0cd7b37e68b2a652959e6572ffa14/lib/core/annotation_src.dart#L156-L158) and [property annotation](https://github.com/angular/angular.dart/blob/f7115aa86da0cd7b37e68b2a652959e6572ffa14/lib/core/annotation_src.dart#L501-L505) that allow for lazy binding to be set up for components mapping (similar to the scope property of the directive definition object in AngularJS v1).

There are at least two issues with this implementation:

1. Overriding default values (this is primarily applicable to components) - when a component is set up with one time bindings from the outer context, it's possible that the initial value in the outer context is not stable until after the initial digest cycle. The main use-case being that the context defines the default value, but watchers or other components bound to this value can change it during the first digest cycle. It is desirable in this case for the one-time binding to be frozen only after the value settles during the digest cycle, which means that the one-time binding is not really a one-time binding but a binding that gets frozen after the first digest.
2. Having two ways (mapping annotation and watch support) to define one time binding is redundant. The mapping annotation is already deprecated and will be removed in the future.

# Existing PRs

* [feat(ngBind): add support for bind-once and lazy bind-once #7367](https://github.com/angular/angular.js/pull/7367)
  + similar to the fast-bind-once approach
* [feat($interpolate): add support for bind-once and lazy bind-once #7366](https://github.com/angular/angular.js/pull/7366)
  + similar to AngularDart's :: approach

Detailed Design

## 

The solution can be split into two parts: simple one-time bindings and labeled bindings.

# Simple one-time bindings

The main purpose of the simple one-time binding is to provide a way to create a binding that gets deregistered and frees up resources once the binding is initialized.

Since we want this binding to work for all possible use cases (interpolation, expressions in directives like ngBind, ngRepeat or even custom directives), it should be implemented as a prefix for the expressions that is identified and processed by Scope's $watch and $watchCollection methods.

When making the decision about the exact time when the binding is automatically deregistered (or "frozen") we need to make a trade off between usefulness and performance.

Eagerly deregistering binding, would make the most sense from the performance perspective but would make this feature unsuitable for cases when data arrives asynchronously after the template instance was created and bound. For this reason, it's not something we are interested in.

Lazy deregistration means that we'll deregister the binding after the binding is set to a non-undefined value. This will consume more resources while data is being fetched, but will make this feature more useful in practice.

There are two issues with lazy deregistration. First one is that the deregistration should occur after the model has settled to avoid the AngularDart issue mentioned earlier (this can be done by deregistering via postDigest queue).

The second issue is that if a binding is never initialized it will keep on using resources. This could be a problem where the model even when loaded contains path that are optional (e.g. employee model has directReports property only if they have reports). In this case the property could we initialized to an empty array or null.

## 

## Simple one-time binding scenarios

The :: syntax used here is not set in stone, but has many great features:

* it's easily visible and distinguishable from the rest of the expression (doesn't look like any existing operator)
* is compatible with both interpolation ( {{foo}} )and expressions expressed literals via attribute values (ng-bind=":: foo")
* works with directives that use micro-syntax ( ng-repeat="user in ::users" )
* can be potentially extended in the future (e.g. to support eager one-time bindings (=:) or labeled bindings (label:)

### simple one-time:

<div>

{{:: user.name}}

</div>

### one-time repeater:

<ul>

<li ng-repeat="user in ::users">

{{user.name}}

</li>

</ul>

### one-time component:

<custom-profile user="::user"></custom-profile>

### one-time repeater + one-time component:

<ul>

<li ng-repeat="user in ::users">

<custom-profile user="::user"></custom-profile>

</li>

</ul>

### one-time repeater + track by:

<ul>

<li ng-repeat="user in ::users track by user.id"> <!-- doesn't make sense -->

{{user.name}}

</li>

</ul>

### repeater + track by + one-time component / one-time binding:

Since track-by allows DOM to be reused with new model, if this DOM is bound using fronzen bindings the DOM will not update. This should be documented as expected behavior.

<ul>

<li ng-repeat="user in users track by user.id">

{{:: user.name }} <!-- will never update -->

<custom-profile user="::user"></custom-profile> <!-- will never update -->

</li>

</ul>

# Labeled bindings

*Note: we currently have no plans to implement this feature unless the issues mentioned below can be resolved.*

Labeled bindings are bindings that have a developer defined label or namespace. Any binding with such label does not participate in regular dirty-checking but instead is processed only when a signal specific to the label occurs in the system.

The main use of this feature would be for bindings that are expensive but change very rarely and when they do change they change as a group. For example translation strings, contents of a large data table that is updated only during pagination.

[fast-bind-on-notify](https://github.com/kseamon/fast-bind/blob/master/src/directives/bind-on-notify/bind-on-notify.js) already addresses most of this use cases but has several downsides:

1. Its functionally is limited only to interpolation like one-way bindings (like ngBind) and thus can't be composed with existing directives (e.g. ngRepeat or custom component directives).
2. The bindings when processed and reprocessed are eager and strictly one-time per signal occurrence which makes them vulnerable in cases when model requires several digest loops to settle.
3. Since the solution depends on Scope events which propagate into isolate scopes by default, we could be accidentally triggering bindings in unrelated isolate scopes.
4. They also execute out of order within the digest cycle but that should generally not be an issue.

All of these issues are an artifact of using events for flushing the bindings.

Alternative approach would baked this functionality into the digest loop. For example, we could have all watches linked via a linked list and digest loop simply traverse this loop (something we already have a PR for), but instead of having just one path through the linked list, we could have a second path containing extra nodes representing labeled bindings. When a signal is fired, the digest loop would start traversing over the longer path as soon as possible and keep traversing the long path the digest is finished. During the traversal over the long path the algorithm would check if the signal name matches the node label and if it does it would dirty check the node, otherwise it would move on.

This would deal with the issues listed above, except for the isolate scope issue, but would significantly complicate things.

Since our long-term goal is to use Object.observe for most of the observation, it makes more sense to forgo implementing this feature and instead focus on making alternative watching mechanism like [watchtower](https://github.com/angular/watchtower.js) which supports O.o() a higher priority.

## Labeled Binding / Region Scenarios

These are just experimental examples to illustrate the scenarios.

### simple labeled binding:

<div binding-notifier="{ u: user }">

{{u: user.name}}

</div>

### labeled binding with repeater:

<div binding-notifier="{ watch: users, notify: 'u' }">

<div ng-repeat="user in u:users">{{u: user.name}}</div>

</div>

# Caveats

## Applicable directives

Ideally, the feature should work for all directives out of the box. At this is the list of core directives that the feature must work against:

* ngBind
* ngBindHtml
* ngBindTemplate
* ngChecked
* ngClass
* ngDisabled
* ngHide
* ngHref
* ngIf
* ngInclude
* ngReadonly
* ngRepeat
* ngSelected
* ngShow
* ngSrc
* ngSrcset
* ngStyle
* ngSwitch
* ngValue

Security Considerations

This feature doesn't change how interpolation or other security-sensitive code in Angular works, so no unusual security considerations are needed.

Performance Considerations / Test Strategy

The primary purpose of this feature is to improve performance. DFA and other production apps that already use custom solutions can verify that the feature works as expected in real world conditions.

Work Breakdown